Lothar Englisch

Gépi kódú programozás a Commodore 64-esen

DATA BECKER – NOVOTRADE

Lothar Englisch

Gépi kódú programozás a Commodore 64-esen

DATA BECKER – NOVOTRADE

A mű eredeti címe: Das Maschinensprache Buch zum Commodore 64 (1984)

Fordította: DOBOSNÉ HARTYÁNYI MÁRIA
Szaklektorok: DR. OBÁDOVICS J. GYULA és DR. LENGYEL JÓZSEF

MÁSODIK KIADÁS

A kiadásért felel: RÉNYI GÁBOR, a NOVOTRADE RT. igazgatója
Kiadványmenedzser: BÉKÉS TAMÁS
Sorozatszerkesztő: ROCHLITZ ANDRÁS
Felelős szerkesztő: TARR KÁLMÁNNÉ
Műszaki szerkesztő: DÉVÉNYI ERIKA

Szedte: az Alföldi Nyomda. Debrecen
Készült a Nyírségi Nyomdában, Nyíregyházán (7,8 A/5 ív)
Budapest, 1986.
ISBN 963 02 4034 3

FONTOS TUDNIVALÓ

A könyvben ismertetett kapcsolások, eljárások és programok nem tekintendők szabadalmi oltalom alá eső ipari termékeknek. Ezek elsősorban amatőr és oktatási célokat szolgálnak. A szerzők rendkívül nagy gondot fordítottak a kapcsolások, műszaki adatok és programok helyességére, a részletek kidolgozása során többszöri ellenőrzést végeztek. Mindez azonban nem zárja ki az esetleges hibalehetőségeket.
Az előforduló hibákért és az ebből adódó következményekért a DATA BECKER cég sem szavatosságot, sem jogi felelősséget nem vállal. Az esetlegesen előforduló hibák közlését a szerzők hálásan fogadják.

ELŐSZÓ

A gépi kódú és assembler szintű programozás különbözőképpen hat a tanulni szándékozókra.
Általában mindenki szívesen megtanulná, sokan megkísérlik, de legtöbben hamar fel is adják, mert a feladat túlságosan bonyolultnak mutatkozik. Nagyon kevesen jutnak el abba az irigylésre méltó helyzetbe, hogy a gépi kódú programozást hatékonyan alkalmazni tudják. Ezzel a könyvvel a Commodore 64-es használóinak egy lehetőség szerint könnyű, járható utat mutatunk a gépi kódú programozás elsajátításához.
A könyv megírására sikerült Lothar Englisch-t megnyerni. Lothar Englisch, aki minden eddig megjelent könyvünk megírásában közreműködött, az egyik legalaposabb ismerője a Commodore operációs rendszerének és az összes Commodore típusú számítógép gépi kódú és assembler programozási nyelvének. Tőle származnak pl. az olyan kényelmes szoftverek, mint a Profi Mon és a Profi Ass.
A jelen előszó írója is többször megkísérelte a gépi kódú programozási nyelv elsajátítását, és mindannyiszor kudarcélményekkel tele hamarosan fel is adta a vállalkozást. Végül ennek a könyvnek a kéziratát áttanulmányozva sikerült eljutnia a célhoz, természetesen azonban ehhez – a könyvön kívül – némi türelemre és kitartásra is szüksége volt. Az tény, hogy a könyv alapos áttanulmányozása nem lesz könnyű dolog, de egyet biztosan állíthatunk: megéri a fáradságot.
Sok örömet kívánunk a könyv olvasásához és sok sikert a gépi kódú programozásban!

Dr. Achim Becker

TARTALOMJEGYZÉK

1. BEVEZETÉS

A gépi kódú programozás előnyei és hátrányai a BASIC-kel szemben

A legtöbb személyi számítógép, így a Commodore 64-es is BASIC nyelven programozható. BASIC nyelven majdnem minden számítógépes feladat megoldható. Ráadásul ez a programnyelv igen könnyen elsajátítható, ezért felmerül a kérdés, hogy egyáltalán miért szükséges a gépi kódú programozással is foglalkoznunk. A továbbiakban összehasonlítjuk a BASIC programozási nyelvet a gépi kódú programozási nyelvvel.
Ebben a könyvben megpróbáljuk bebizonyítani, hogy a gépi kódú programozási nyelvet éppen olyan könnyen meg lehet tanulni, mint a BASIC-et. Természetesen sokat segít, ha az olvasó a BASIC nyelvet alaposan ismeri. A gépi kódú nyelv logikája némileg eltér a BASIC-étől.
A Commodore 64-es BASIC nyelve rövidített változata a *B*eginners *A*ll *P*urpose *S*ymbolic *I*nstruction *C*ode (kezdők számára készült, általános célú, szimbolikus utasításokból álló kódrendszer) programozási nyelvnek. A BASIC nyelv az úgynevezett magas szintű programozási nyelvek közé tartozik, mint például a FORTRAN, a PASCAL vagy a COBOL. Ezeket a nyelveket problémaorientált nyelveknek hívjuk, mivel egy-egy feladattípus (matematikai vagy pl. ügyviteli számítások) megoldására dolgozták ki őket. Szemben az úgynevezett géporientált nyelvekkel, mint például a FORTH, melyek elsősorban a számítógép hardverfelépítéséhez igazodnak. A géporientált nyelvek közé tartozik a processzorok saját nyelve, a gépi kódú nyelv is.
A magas szintű programnyelveket a számítógépek nem „értik" meg. Joggal kérdezhetjük, hogy ennek ellenére miért tudja a gép mégis olyan gyorsan végrehajtani a BASIC parancsokat. Erről a Commodore 64 BASIC interpreterje (értelmező) gondoskodik. Az általunk megírt program végrehajtásakor az interpreter minden egyes parancsot egyenként értelmez, vagy idegen szóval interpretál. Ezt a folyamatot a programozónak tulajdonképpen nem kell tudomásul vennie, hiszen ez a rendszer „belső" ügye.
Gépeljük be például:

```
PRINT "HALLO"
```

majd nyomjuk meg a RETURN billentyűt. Az interpreter elolvassa a parancsunkat jelről jelre. Az első szót végigolvasva, összehasonlítja azt az utasításkészletében elhelyezkedő szavakkal (GOTO, FOR, INPUT). Megvizsgálja, hogy az általunk begépelt szó szerepel-e az utasításkészletében, és ha igen, megjegyzi, hogy hányadik helyen. A sorszámra azért lesz szüksége, hogy az interpreteren belül a megfelelő parancs helyzetét megállapíthassa. Most kerülhet sor a PRINT parancs végrehajtására. Az interpreter most ismét jelről-jelre továbbhalad, az idézőjel elolvasásakor tudomásul veszi, hogy egy szöveg kinyomtatása következik, és megkeresi az idézőjel párját. Továbbhaladva azt vizsgálja, hogy van-e még egyéb szöveg is az idézőjel után, ha nincs, akkor a parancsot végrehajtja és megjelenik a READY üzenet.

A gépi kódú program végrehajtási sebessége

Mint azt az előzőekben láttuk, a BASIC parancsokat a processzor nem tudja közvetlenül végrehajtani. Végrehajtás előtt az interpreter megkeresi a parancsot, majd előkészíti a processzor számára, és ez nyilván időbe telik.
A POKE 1024,10 BASIC parancs végrehajtása az értelmezéssel együtt kb. 2 millisecundumot vesz igénybe. Ezzel a BASIC paranccsal az 1024-es tárcímre 10-et írunk. A feladat gépi nyelven két utasítással írható le:

```
LDA #10
STA 1024
```

Ezek végrehajtásához a processzornak kb. 6 mikrosecundumra, azaz 300-szor kevesebb időre van szüksége, mint a BASIC parancs végrehajtásához. Azonos feladatoknál a gépi kódú program végrehajtási ideje tized-, sőt ezredrésze is lehet a BASIC program végrehajtási idejének. A végrehajtási sebesség különösen fontos a számolásigényes matematikai feladatok megoldásánál, illetve az adathalmazok rendezésénél.
Nagy adathalmazok esetén a rendezés BASIC nyelven megírt programmal több óráig is tarthat. Így ezeket a feladatokat gépi kódú nyelv nélkül gyakorlatilag lehetetlen megoldani. Bizonyos feladatokhoz azért írunk gépi kódú programot, hogy gépidőt takarítsunk meg, de vannak olyan feladattípusok is, amelyeket csak gépi kódban írt programmal oldhatunk meg. Ide tartoznak a megszakítási technikát igénylő feladatok, amelyek a külső egységek közötti, vagy a gép és egy külső egység közötti adatforgalmat bonyolítják le. A megszakítási technika lényege az, hogy egy külső egység képes a gép éppen folyamatban lévő munkáját egy bizonyos ideig felfüggeszteni, és a gépet arra kényszeríteni, hogy az ő igényeit kiszolgálja. Általánosságban elmondhatjuk, hogy a gépi kódú nyelv használata nélkül a Commodore 64-es személyi számítógép összes beépített lehetőségét nem tudjuk kihasználni és természetesen ugyanez vonatkozik bármely személyi számítógépre. Különösen igaz ez a grafika és a hangszintetizátor programozására.
A gépi kódú programozási nyelvek további előnye a tárolókapacitás takarékos kihasználása. Egy jól megírt gépi kódú program, összevetve az azonos feladatra megírt BASIC nyelvű programmal tized annyi helyen elfér a tárban. Az Olvasó biztosan tisztában van azzal, hogy egy 1 kbyte-os BASIC program nem nevezhető nagyméretű programnak. Ugyanakkor egy 1 kbyte-os gépi kódú program már tekintélyes feladatot takar.
A takarékos helykihasználás az adatok tárolására is vonatkozik. Az adatokat csak gépi kódú programmal lehet tökéletesen összesűrítve elhelyezni. Ha például egy olyan táblázatot tárolunk, amelynek minden eleme nulla és száz közé eső egész szám, BASIC programmal dolgozva elemenként minimum két byte-ra van szükségünk. A BASIC nyelvben ugyanis a legkisebb helyigényű változó típust, az egész számot az interpreter két byte-on tárolja, holott a nulla és száz közé eső egész számok egy byte-on is elférnének. Minthogy gépi kódú programmal az egy byte-os tárolás megoldható, ugyanaz az adathalmaz fele annyi helyet foglal el a tárban gépi kódú programmal dolgozva, mint a BASIC program esetében.
A gépi kódú nyelven megírt programok tökéletesen képesek igazodni a feldolgozandó adathalmaz szerkezetéhez.
Sok előnyös tulajdonsága mellett a gépi kódú programozásnak is vannak hátrányai. A „gépi kódú programozás" elnevezés egy kicsit megtévesztő. A „gépi kód" kifejezés ugyanis valójában az egyes műveletek számszerű kódjára utal, hiszen a processzor kizárólag bináris alakú számkódokkal tud dolgozni. Ha a programozónak minden műveletet ilyen bináris kódokkal kellene leírnia, valószínűleg visszariadna a feladattól. Ahhoz, hogy a BASIC programozáshoz

hasonló könnyedséggel programozhassunk gépi kódban is, szükség van egy segédnyelvre, ami átmenet a tényleges gépi kód és a magas szintű programnyelvek között. Ez az átmeneti nyelv az assembler nyelv. Ebben a könyvben található egy ASSEMBLER nevű program, amellyel assembler nyelvű programokat szerkeszthetünk. Az assembler nyelv abban különbözik a tényleges gépi kódú nyelvtől, hogy míg az utóbbiban az egyes utasításokat számkódok formájában, addig az előbbiben olyan szöveges szimbólumok formájában adhatjuk meg, amelyek hasonlóan a BASIC szavakhoz emlékeztetnek arra a műveletre, amelyre az utasítás vonatkozik.

A gépi kódú programozási nyelv egyik legnagyobb hátránya, a processzororientáltság. A programokat csak arra a gépre tudjuk átvinni, amely ugyanazzal a processzortípussal rendelkezik, mint amelyen a programot megírtuk. A gépi nyelv másik hátránya a BASIC nyelvhez képest, hogy a gépi kódú nyelven megírt program tesztelése meglehetősen nehézkes. A gépi kódú programok tesztelésének megkönnyítésére közlünk egy szimulátor programot, amellyel követhetjük a programlépéseket.

Összefoglalva azt mondhatjuk, hogy a gépi kódú programozásnak megvan a létjogosultsága. Sok feladat csak gépi nyelven oldható meg, továbbá csak így tudjuk kihasználni számítógépünk minden beépített lehetőségét. Gyakorlott programozók legtöbbször a főprogramot BASIC nyelven, a kritikus részeket pedig gépi kódban írják meg. Ebben a könyvben ismertetjük azokat a módszereket, amelyek lehetővé teszik, hogy gépi nyelven megírt rutinokat könnyedén és kényelmesen felhasználhassunk egy BASIC nyelven megírt programmal együtt.

2. A 6510-es mikroprocesszor

Mielőtt megismerkednénk a gépi kódú programozási nyelvvel, néhány szóban ismertetjük magát a processzort, továbbá néhány alapfogalmat, mint pl. regiszter, adatok, címek, bit, byte stb. Először vizsgáljuk meg közelebbről a mikroprocesszor belső felépítését.

A 6510-es mikroprocesszor a 65XX processzorok családba tartozik. Ezeket a processzorokat találjuk a Commodore gépcsalád minden tagjában. A processzor ún. *regiszter*eket tartalmaz, amelyekben a műveletek lezajlanak. Mik is tulajdonképpen ezek a regiszterek?

A processzor digitális elven működik, azaz mindössze két állapotot tud egymástól megkülönböztetni, egy bekapcsolási (BE), ill. egy kikapcsolási (KI) állapotot.

A két állapot jelölésére használt 1-et, ill. 0-t, a továbbiakban bitnek nevezzük (a **bi**nary digi**t** angol szavakból). Természetesen egy bit önmagában kevés, ezért a processzor nyolc összekapcsolt bitből álló, vagy másképpen egy byte-os regiszterekkel dolgozik.

a bit száma	7	6	5	4	3	2	1	0
tartalma	0	1	1	0	1	0	0	1

Az ábrán megszámoztuk a biteket 0-tól felfelé 7-ig. Hogyan számíthatjuk ki a nyolc biten, azaz egy byte-on tárolt bináris szám értékét?

Vizsgáljuk meg először közelebbről a decimális számokat:

decimális hely	3	2	1	0	(10 hatványa)
tartalom	5	7	2	4	

Most is megszámoztuk az egyes helyeket, 0-tól 3-ig. Hogyan kapjuk meg a decimális szám értékét? Minden decimális számjegy 0 és 9 közé esik, és minden helyiérték az előtte álló helyiérték tízszerese.

$$4+2\times10+7\times10\times10+5\times10\times10\times10=5724$$

A decimális számokhoz hasonlóan számíthatjuk ki a regiszterek tartalmának megfelelő számértéket is azzal a különbséggel, hogy a bináris számjegyek értéke csak 0 vagy 1 lehet, ill. minden helyiérték az előző helyiérték kétszerese. A fenti regiszter tartalmának megfelelő számérték tehát:

$$1\times2^0+0\times2+0\times2\times2+1\times2\times2\times2+0\times2\times2\times2\times2+1\times2\times2\times2\times2\times2+1\times2\times2\times2\times2\times2\times2$$
$$+0\times2\times2\times2\times\ \ \times2\times2\times2\times2=1+0+0+8+0+32+64+0=105$$

A fenti számítást egyszerűbben is elvégezhetjük, ha előre meghatározzuk az egyes helyiértékek számértékét:

Helyiérték	**Számérték**
0	2 ↑ 0=1
1	2 ↑ 1=2
2	2 ↑ 2=4
3	2 ↑ 3=8
4	2 ↑ 4=16
5	2 ↑ 5=32
6	2 ↑ 2=64
7	2 ↑ 7=128

Számítsuk ki, mekkora lehet maximálisan a regiszter tartalmának számértéke. A legnagyobb értéket nyilván akkor kapjuk, ha minden helyiértéken 1 áll, és ekkor az eredmény 1+2+4+8+16+32+64+128=255. A nyolc biten kifejezhető legnagyobb decimális szám tehát 255. Így nyolc biten összesen 256 (0-tól 255-ig) különböző számot ábrázolhatunk. Ezt az értékkészletet feltehetően már ismeri az olvasó a BASIC PEEK utasítással kapcsolatban. Minthogy a kettes számrendszerbeli ábrázolásmód rendkívül körülményes, a továbbiakban hexadecimális számrendszerben fogunk dolgozni. Egy bineáris szám leírva is igen nagy terjedelmű, ami kényelmetlen lehet a programozás során. Ha azonban a nyolc bites bináris számot felosztjuk két négy bites számra, mindkét négy bites egység 16 különböző értéket vehet fel. Minthogy a 16-os vagy másképpen hexadecimális számrendszerben éppen 16 különböző számjegy van, minden nyolc bites bináris szám két 16-os számrendszerbeli számjegynek felel meg.

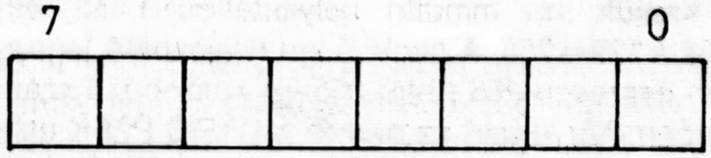
bit száma	7	6	5	4	3	2	1	0
bináris tartalom	0	1	1	0	1	0	0	1
hexadecimális tartalom		6				9		

Minden byte-ot felosztunk két fél byte-ra, melyeket az irodalomban gyakran *nybble*-nek (falatnak) is neveznek. Minden nybble 16 különböző értéket vehet fel (0-tól 15-ig). A következő táblázat a bináris, hexadecimális és decimális számok közötti kapcsolatot mutatja. A fenti regiszter tartalma a hexadecimális 69. Hogy a továbbiakban a különböző számrendszerekbeli jelölést meg tudjuk különböztetni egymástól, a hexadecimális számoknál a megszokott $ jelet, míg a bináris számok mellett egy % jelet fogunk használni. A későbbiekben főként hexadecimális számokkal dolgozunk, mivel egyrészt közel áll a processzor bináris logikájához, másrészt hexadecimális rendszerben egy nyolc bites érték két számjeggyel kifejezhető.

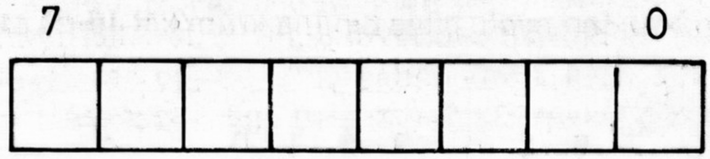
Bináris	**Hexadecimális**	**Decimális**
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0105	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Tekintsük végig a processzor legfontosabb regisztereit. A processzor összesen hat regisztert tartalmaz. Ezek között van 5 nyolc és 1 tizenhat bites regiszter. A regiszterek közül az *akkumulátor* a legfontosabb. Az akkumulátor a processzor általános munkaregisztere, amelyet a processzor minden aritmetikai, logikai, illetve összehasonlító művelet végrehajtása során igénybe vesz.

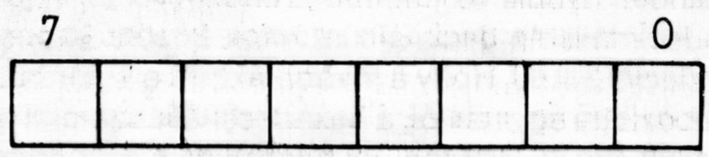
Akkumulátor



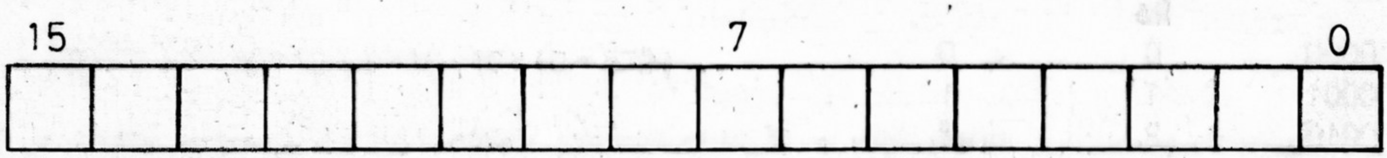
X-Regiszter



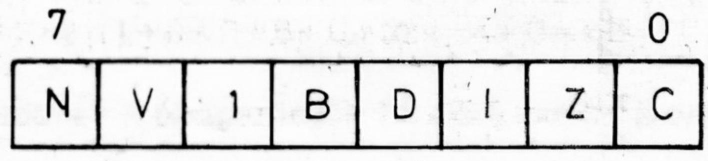
Y-Regiszter



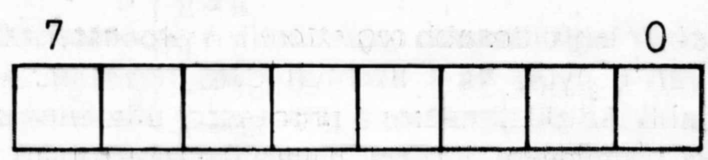
Utasításszámláló



Veremregiszter



Állapotregiszter



A processzor második regisztere az *X regiszter*. Táblázatok vagy mátrixok feldolgozásánál ez a regiszter együttműködik az akkumulátorral olyan számlálóként, amely az egyes táblázatelemekre mutat. Ezt a regisztert ezért indexregiszternek is nevezzük.

Az *Y regiszter* ugyanazt a szerepet tölti be, mint az X regiszter és ugyanazt a célt is szolgálja.

Az *utasításszámláló* egy 16 bites regiszter. Tartalma az a tárcím, ahol a soron következő végrehajtandó utasítás elhelyezkedik. Ezt a regisztert közvetlenül a mikroprocesszor irányítja. Tartalmához általában nem férhetünk hozzá.

Az úgynevezett *veremregisztert* az alprogramok használják átmeneti tárolóhelyként. Ennek jelentőségét majd a szubrutinok vagy alprogramok programozásánál fogjuk megismerni.

Végül az *állapotregiszter* felvilágosítást ad az utolsó végrehajtott parancs eredményéről, alapját képezi a döntéseknek és a feltételes utasításoknak. Az állapotregiszter nyolc bitje közül hét úgynevezett *kapcsoló* szerepet tölt be. Ezek a kapcsolók (flagek) közvetlenül lekérdezhetőek. Ha például egy elágazó utasítás csak bizonyos feltétel esetén következik be, a feltétel bekövetkezését egy kapcsoló 0, ill. 1 állapota jelezheti. Az állapotregiszter felépítését a következő ábra mutatja:

7	6	5	4	3	2	1	0
N	V	–	B	D	I	Z	C

Az ábrán a számok alatt található betűk az egyes kapcsolók neveinek rövidítései. Jelentésük a következő:

C – a *CARRY* (átviteli) kapcsoló, amely azt mutatja, hogy egy műveletnél történt-e átvitel vagy sem. Két szám összeadásakor, ha az eredmény nagyobb, mint 255, és így nyolc biten nem tárolható, a CARRY kapcsoló értéke 1 lesz.

Z – a *ZÉRÓ* kapcsoló, amelynek értéke 1, ha valamely művelet eredménye 0.

I – az *INTERRUPT* (megszakítás) letiltás kapcsoló megmutatja, hogy a program végrehajtása során a programmegszakítás megengedett-e vagy sem.

D – a *DECIMÁLIS* kapcsoló, amely 1 értéket vesz fel, ha az összeadás vagy kivonás decimális alakban történik.

B – a *BREAK* kapcsoló a BRK utasítás által okozott megszakításra utal.

V – az *OVERFLOW* (túlcsordulás) kapcsolót akkor használjuk, ha előjeles számokkal dolgozunk.

N – a *NEGATÍV* kapcsoló értéke mindig 1, valahányszor a művelet eredménye nagyobb, mint 127. A kapcsoló neve arra utal, hogy minden érték, amely $7F felett van, negatív számként értelmeződik.

Ahhoz, hogy a processzor dolgozni tudjon, egy programot, és a program végrehajtásához szükséges adatokat kell tartalmaznia. Mindezek elhelyezése a tárban történik, amely a regiszterekhez hasonló nyolc bites egységekre, *rekeszekre* van felosztva. A program végrehajtásakor a processzor az utasításokat és az adatokat valamelyik rekeszben keresi. Az aktuális rekesz kiválasztása a tár címzése alapján történik. A tár minden egyes rekesze egy meghatározott címmel rendelkezik, amely tulajdonképpen egy bináris szám. Ha a processzor a címeket is csak nyolc biten tárolná, mindössze 256 tárcímet (0-tól 255-ig) tudna megkülönböztetni, ami természetesen nem elég. A gép tizenhat bites címekkel dolgozik, azaz a processzor a nyolc bites adatbusszal szemben tizenhat bites címbusszal van ellátva.
Pontosabban: a 6510-es processzor 65536 tároló rekesszel rendelkezik, melyek mindegyike 0–255 közötti értéket tartalmazhat. A kényelmesebb kezelés miatt $2^{10} = 1024$ byte-ot 1 kilobyte-nak vagy 1 kbyte-nak nevezzük. A Commodore 64-es tehát 64*1024 = 65536 byte-ot vagyis 64 kbyte-ot tud címezni.
Mindezek alapján érthető, hogy a tizenhat bites ún. program- vagy utasításszámláló regiszter milyen fontos szerepet tölt be a processzor munkájában. Ez a regiszter tartalmazza ugyanis mindig a soron következő, végrehajtandó utasítás tárbeli címét. A mikroprocesszor számára maga az utasítás is egy 0–255 közé eső szám, amely 256 utasítás (parancs) megkülönböztetését teszi lehetővé.
A 6510-es esetében nem minden kód (0–255 között) takar utasítást, mivel az utasítások száma kevesebb mint 256.
Természetesen ezek nem BASIC, hanem a processzor által azonnal végrehajtható gépi kódú utasítások.

3. A 6510-ES UTASÍTÁSOK ÉS CÍMZÉSMÓDOK

A nyolc biten megkülönböztethető 256 kód közül 151-nek van tényleges jelentése, azaz a 6510-es processzor utasításkészlete 151 különböző utasításból áll. A 151 utasítás között sok olyan van, amelyek csak a címzésmódban különböznek egymástól.
A 6510-es processzor csak 59 teljesen különböző utasítást használ. Az utasításszavakat nagyon könnyű megjegyezni. A következőkben bemutatjuk az utasításcsoportokat, és azon belül ismertetjük a lehetséges címzésmódokat is.

3.1 A betöltés

Az utasítás célja egy megadott tárcím tartalmának betöltése egy regiszterbe. Mivel három munkaregiszter van, háromféle betöltőutasítást különböztetünk meg.

LDA	betöltés az akkumulátorba
LDX	betöltés az X regiszterbe
LDY	betöltés az Y regiszterbe

A fenti utasítások használatához meg kell ismerkednünk a címzésmódokkal, amelyek meghatározzák azt az eljárást, amellyel a processzor az adatot tartalmazó rekesz címét megkapja.

A közvetlen címzés

```
LDA #10
```

A közvetlen címzésmódot a # jel különbözteti meg a többitől.
Jelentése: a 10-et mint számértékét be kell tölteni az akkumulátorba.
(A megfelelő BASIC parancs: A=10)
Ezt a címzésmódot használhatjuk, ha az akkumulátorba egy konstans számértéket kívánunk tölteni.
A közvetlen címzés az X, ill. Y regiszterekre is használatos:

```
LDX #$7F        LDY #$AB
```

A fenti két utasítással az X regiszterbe 127-et, az Y regiszterbe 171-et töltünk.
A betöltött számérték éppúgy a program része, mint a BASIC programban. Az utasítás kódja és a számérték két egymást követő tárcímen helyezkedik el. Ha egy gépi kódú programot erről a címről elindítunk, a processzor a rekesz tartalmát mint utasítást értelmezi. Ha az elhelyezett érték $A9, vagy decimálisan 169, a gép tudja, hogy ez az LDA utasítást jelenti, és így a következő tárcím tartalmát betölti az akkumulátorba. Az utasítás tehát két byte-ot foglal el, ezért a végrehajtás után a programszámláló értéke automatikusan kettővel nő.

Az abszolút címzés

Ha egy regiszterbe nem konstans értéket, hanem egy tárcím tartalmát akarjuk betölteni, abszolút címzést használhatunk.

```
LDA $C0AF
```

A $C0AF tárcím tartalmát betöltjük az akkumulátorba. A $C0AF cím egy 16 bites szám, egy rekeszben azonban csak nyolc bit fér el, ezért a 16 bites számot két nyolc bites részre kell felbontani. A tárban az utasítás kódja után következik a cím alsó, majd felső byte-ja. Az egymást követő rekeszek tartalma az utasításkód, $AD (173), majd $AF (175), végül $C0 (192). A megfelelő BASIC utasítás:

```
A=PEEK($C0AF)
```

Hasonló utasításokat használhatunk az X, illetve az Y regiszterek töltésére is. Az utasításkódok megtalálhatóak a függelékben.
Az utasítás végrehajtásakor a processzor „tudja", hogy abszolút címzésről van szó, így az utasításkódot tartalmazó rekeszt követő két rekesz tartalmát a tárcím alsó, ill. felső byte-jának tekinti.
Végrehajtás után az utasításszámláló tartalma hárommal nő. A három byte-os utasításokon kívül a 6510-es processzor két byte-os, sőt egy byte-os úgynevezett operandus nélküli utasításokkal is dolgozik.
Az állapot (vagy státusz) regiszter fontos szerepet játszik az LD utasítás végrehajtásakor. Ha a betöltött érték 0, a Z kapcsoló 1, egyébként 0 lesz. Ha a betöltött érték negatív, azaz nagyobb mint $7F (127), az N kapcsolóba 1, egyébként 0 kerül.

A nulláslap címzés

Ez a címzésmód a 65XX processzorok sajátossága. Akkor használhatjuk, ha egy cím 0 és $FF (255) közé esik. Ekkor ugyanis a cím kifejezhető nyolc bittel, és így egy három byte-os utasítás tárolásához is elegendő két byte. Azon túl, hogy ez tárhely-megtakarítás, a végrehajtás is gyorsabb. A címzésmód elnevezése onnan ered, hogy a teljes tár 256, egyenként 0–255 címet tartalmazó ún. lapra van felosztva.

A betöltési parancs ekkor:

```
LDA $73
```

Tárolása két byte-on történik: $A5 (165), $73 (115).

BASIC-ben:

```
A=PEEK($73)
```

Az indexelt címzés

Az indexelt címzésben az X és Y regiszterek fontos szerepet töltenek be.

```
LDA $25B8,X
```

Ez az úgynevezett X-szel indexelt abszolút címzés. A processzor az akkumulátorba nem a $25B8 rekesz tartalmát tölti be, hanem ehhez előbb hozzáadja az X regiszter tartalmát. Ha például az X regiszter tartalma $35, akkor a processzor elvégzi az alábbi összeadást:

```
$25B8+$35 = $25ED
```

majd betölti az akkumulátorba a $25ED rekesz tartalmát. Ez a címzésmód igen hasznos ciklusok programozásánál, táblázatok feldolgozásánál. A későbbiekben erre még sok példát látunk. Ugyanez az utasítás BASIC-ben:

```
A=PEEK($25B8+X)
```

ahol X az X regiszter tartalmát jelöli. Természetesen az X indexregiszter helyett az Y-t is használhatjuk. Például:

```
LDA $25B8,Y
```

Így két független ciklusváltozót kezelhetünk, amire pl. egymásba ágyazott ciklusok programozásakor van szükség. Az indexelt címzést nulláslap címzéssel együtt is használhatjuk. Például:

```
LDA $BA,X
```

Az indirekt indexelt címzés

Az indirekt indexelt címzés talán a legnehezebben érthető, de igen hasznos címzésmód. Ekkor ismét fontos szerepet tölt be a nulláslap. Az indirekt indexelt címzésnél a nulláslap két egymást követő rekesze egy mutatót képez a kívánt címre. Az első byte a cím alsó, a második byte a cím felső byte-ja.
Tegyük fel, hogy a nulláslap $70-es címének tartalma $20, a $71-es tartalma $C8. Ekkor a két cím együtt a $C820 címet adja. Ahhoz, hogy a tényleges címet megkapjuk, az előző címet az Y regiszter tartalmával indexeljük. Ha pl. az Y-ban $B3 van, ez még hozzáadódik a $C820 értékhez.

```
LDA ($70), Y   ($70)⇒$20
               ($71)⇒$C8
                     $C820
                 (Y)⇒$B3
                     $C8D3
             ($C8D3)⇒$4F
A=$4F
```

A fentiekkel egyenértékű BASIC utasítás:

```
A=PEEK(PEEK($70)+256•PEEK($71)+Y)
```

Az indirekt címzést formailag arról lehet megismerni, hogy az operandusa zárójelben van. A címzésmód nagyon hatékony: egy 2 byte-os utasítással egy teljes tárterületre utalhatunk. Rugalmasabb, mint az egyszerű indexelt címzés, mivel itt nemcsak egy lapot kezelhetünk, hanem valóban az egész tárterületet, és eközben csak a nulláslap egy 2 byte-os mutatóját kell változtatnunk.

Az indexelt indirekt címzés

Ellentétben az indirekt indexelt címzéssel, most nem az Y, hanem az X regiszterrel dolgozunk. Itt is a nulláslap két egymást követő címén képezzük a tényleges címet. A kapott mutatókhoz hozzáadjuk az Y index tartalmát, és végül az így kapott cím tartalmát tekintjük a művelet operandusának.
Nézzünk erre is egy példát:

```
LDA ($70,X)       (X)=>$08
               =>($78)=>$40
                 ($79)=>$20
                       $2040
               ($2040)=>$A9
A=$A9
```

Ugyanez BASIC utasítással:

```
A=PEEK(PEEK($70+X)+256•PEEK($70+X+1))
```

A nulláslap egymást követő két címéhez hozzáadjuk az X regiszter tartalmát, majd az így kapott két értéket egy további tárcím alsó és felső byte-jának tekintjük, végül a tárcím tartalmát betöltjük az akkumulátorba.
Az indexelt indirekt címzésmódot az indirekt indexelt címzésmódhoz képest viszonylag ritkábban használjuk.

Foglaljuk össze az eddig megismert címzésmódokat és utasításkódokat:

Címzésmód	LDA	LDX	LDY
közvetlen	$A9	$A2	$A0
abszolút	$AD	$AE	$AC
nulláslap	$A5	$A6	$A4
abszolút, X-szel indexelt	$BD	–	$BC
abszolút, Y-nal indexelt	$B9	$BE	–
nulláslap, X-szel indexelt	$B5	–	$B4
nulláslap, Y-nal indexelt	–	$B6	–
indirekt indexelt	$B1	–	–
indexelt indirekt	$A1		

A további címzésmódokkal, a relatív címzéssel és az akkumulátorcímzéssel majd a megfelelő utasítások tárgyalásakor foglalkozunk.

3.2 A tárolóutasítások

A betöltőprogram utasítások ellentétei a tárolóutasítások. A tárolóutasításokkal megváltoztathatjuk egy adott tárcím tartalmát:

STA
STX
STY

Az A (akkumulátor), az X és az Y regiszterek tartalma betöltődik az operandus által megadott tárcímre. Ezeknél az utasításoknál ugyanazok a címzésmódok állnak rendelkezésre, mint az LD (betöltési) utasításoknál, kivéve a közvetlen címzést, itt ugyanis mindig meg kell adni azt a címet, ahova a regiszter tartalmát be kell írni. Mivel tárolás közben a regiszterek tartalma nem változik, ezek az utasítások egyetlen kapcsoló értékét sem befolyásolják.

Címzésmódok és az utasítások kódjai:

Címzésmód	STA	STX	STY
abszolút	$8D	$8E	$8C
nulláslap	$85	$86	$84
abszolút, X-szel indexelt	$9D	–	–
abszolút, Y-nal indexelt	$99	–	–
nulláslap, X-szel indexelt	$95	–	$94
nulláslap, Y-nal indexelt	–	$96	–
indirekt indexelt	$91	–	–
indexelt indirekt	$81	–	–

A megfelelő BASIC utasítást az Olvasó biztosan ismeri: BASIC nyelvben a POKE utasítással írhatunk be tetszőleges értéket egy meghatározott tárcímre:

STA $8000	POKE $8000,A
STX $C020,Y	POKE $C020+Y,X
STY $F1	POKE $F1,Y

A tárparancsok a címzésmódtól függően két, ill. három byte-ot foglalnak el a tárban.
A betöltés és a tárolás utasításai (LD, ST) a gépi kódú nyelv legalapvetőbb részét képezik, ezek biztosítják ugyanis a regiszterek és a tár közötti kommunikációt.

3.3 A processzor belső átviteli utasításai

Az átviteli utasítások valamely regiszter tartalmát egy másik regiszterbe töltik, például az X regiszter tartalmát az akkumulátorba, vagy megfordítva. Az átviteli utasításoknak azért kell különös jelentőséget tulajdonítanunk, mert a legtöbb utasítás csak az akkumulátor tartalmát módosítja. Átvitel közben a forrásregiszter tartalma nem változik meg.
Közvetlen átvitel az X, ill. Y regiszterek tartalma között ezekkel az utasításokkal sem lehetséges. Az ilyen átvitelt az akkumulátoron keresztül kell lebonyolítani.
Minden belső átviteli utasítás 1 byte hosszú, hiszen operandusra itt nincs szükség. A műveletet az utasításkód egyértelműen tartalmazza, meghatározza. Nézzünk néhány példát a belső átviteli utasításokra, a megfelelő BASIC utasításokkal együtt:

```
TAX          X=A
```

Az akkumulátor tartalmát az X regiszterbe másoljuk. A Z és N kapcsolók módosulhatnak, az akkumulátor tartalma változatlan marad.

```
TXA          A=X
```

Hasonló utasítások az Y regiszterre:

```
TAY          Y=A
TYA          A=Y
```

A következő két utasítás a veremmutatóval dolgozik, melynek tartalma csak az X tartalmával cserélhet helyet.

```
TSX          X=SP
```

A veremmutató tartalma betöltődik az X regiszterbe. A Z és N kapcsolók tartalma a betöltött értéknek megfelelően módosulhat. A veremmutató tartalma változatlan marad.

```
TXS          SP=X
```

Az előző utasítás megfordítottja. A kapcsolók nem változnak, mivel a veremmutató nem munkaregisztere a processzornak.

Az átviteli utasítások kódjai

TAX	$AA
TXA	$8A
TAY	$A8
TYA	$98
TSX	$BA
TXS	$9A

3.4 Az aritmetikai utasítások

A 6510-es processzor az összeadás és a kivonás műveleteket a következőképpen végzi el: Minden műveletnek két operandusa van; a processzor az első operandust az akkumulátorban keresi, a másodikat egy tárcímről tölti be. Az eredmény mindig az akkumulátorban képződik. Hasonló elven működnek a logikai műveletek is, melyeket a későbbiekben fogunk tárgyalni.
Az aritmetikai műveletek közül tekintsük először az összeadást: a megadott tárcím tartalma hozzáadódik az akkumulátor tartalmához, és az eredmény ismét az akkumulátorba kerül.

```
ADC #$3A          A=A+$3A
```

Ha két nyolc bites értéket kell összeadni, előfordulhat, hogy az eredmény nem fejezhető ki nyolc bittel, azaz átvitel keletkezik.
Nézzünk először egy példát a bináris összeadásra. A végrehajtandó művelet ADC #$3A, miközben az akkumulátor tartalma $9E.

```
$9E=%10011110
$3A=%00111010
```

A bináris összeadás művelete:

```
  10011110
 +00111010
 ---------
  11011000=$D8
```

A bináris összeadást a decimális összeadáshoz hasonlóan kell elvégezni. A számjegyek összeadása során négy esetet különböztetünk meg:

0+0=0
0+1=1
1+0=1
1+1=0+átvitel

Az átvitel szerepe ugyanaz, mint a decimális összeadásnál.

A fenti példában a végeredmény ($D8) még elfér nyolc biten, a következő példában azonban már gondoskodni kell az átvitelről. A végrehajtandó művelet:
ADC #$3A, miközben az akkumulátor tartalma $E4.

```
$E4 %11100100
$3A %00111010
```

A bináris összeadás:

```
  11100100
 +00111010
 ---------
  100011110=$11E
```

Az eredmény nem fér el a nyolc bites akkumulátorban, a gép a CARRY átviteli kapcsolót használja. Az ilyen összeadásoknál az átvitelt a C kapcsoló jelzi, azaz értée 0, ha nem volt átvitel, és 1 ha volt, ezért ezt a kapcsolót az akkumulátor kilencedik bitjének is nevezzük.
Az átviteli kapcsoló beiktatásával olyan számok is összeadhatóak, amelyek nyolc biten nem férnek el. Ha a számok tárolására kétszer nyolc bitet használunk, az ábrázolható számtartomány a 0-tól 65535-ig terjedő intervallumra bővül. Két ilyen szám összeadásakor a gép először összeadja az alsó nyolc bitet, megjegyzi az átvitelt, majd folytatja az összeadást a felső nyolc bittel.
Minden ilyen összeadás előtt természetesen törölni kell az átviteli kapcsolót.
A gépi kódú utasítás, és BASIC megfelelője:

ADC #$3A   A=A+$3A+C

A művelet elvégzésekor az N és Z kapcsoló értéke is megváltozhat, ha az összeadás eredménye nulla vagy negatív (ha az eredmény hetedik bitje 1) szám. Az összeadási művelet a V kapcsolót is érinti, de mivel ez csak ez előjeles összeadásnál használatos, jelentésével csak a későbbiekben foglalkozunk.
A következő táblázat tartalmazza az ADC utasítások kódjait:

Címzésmódok	ADC
közvetlen	$69
abszolút	$6D
nulláslap	$65
abszolút, X-szel indexelt	$7D
abszolút, Y-nal indexelt	$79
nulláslap, X-szel indexelt	$75
indirekt indexelt	$71
indexelt indirekt	$61

A processzor az összeadáshoz hasonlóan végzi el a kivonást. Az akkumulátor tartalmából kivonja a megadott című byte tartalmát, és az eredményt az akkumulátorba tölti. Ha a kivonásban szereplő számok kívül esnek a 0-tól 255-ig terjedő számtartományon, a művelet elvégzésekor alulcsordulás következhet be (pl. ha az eredmény kisebb mint nulla). Az alulcsordulást is a C kapcsoló jelzi. Ha nem volt alulcsordulás, a C kapcsoló értéke 1 lesz. Többhelyiértékes kivonás előtt a C kapcsoló értékét 1-re kell állítani, jelezve, hogy még nem volt alulcsordulás.
Például:

SBC #$3A   A=A−$3A−(1−C)

A bináris kivonás műveleti szabályai hasonlóak az összeadás műveleti szabályaihoz.
A négy különböző eset:

0−0=0
0−1=1+alulcsorduló átvitel
1−0=1
1−1=0

Ha az akkumulátor tartalma $7F, a két szám bináris alakja:

```
$7F = %01111111
$3A = %00111010
```

A bináris kivonás:

```
  01111111
 -00111010
 ---------
  01000101
```

A kivonás eredménye %01000101, azaz hexadecimálisan $45. Mivel alulcsordulás nem volt, a C kapcsoló értéke 1 marad.

Nézzük meg, mi történik, ha az akkumulátor tartalma $1E. A bináris kivonás:

```
  00011110
 -00111010
 ---------
  11100100
```

A kivonás eredménye %11100100, hexadecimálisan $E4. Mivel azonban átvitel történt, a C kapcsoló értéke 0 lesz.
Decimálisan a fenti művelet a következő:

$30 - 58 = -28$

Hogyan lehet a kapott bináris eredményt értelmezni? Az eredmény $E4, decimálisan 228, ezt kivonva 256-ból a tényleges eredményt kapjuk. A C = 0 arra utal, hogy az eredményt negatív számként kell értelmezni. A negatív számok ábrázolása az ún. *kettes komplemens*ükkel történik.
Egy bináris szám kettes komplemensét megkapjuk, ha minden bitjét ellenkezőjére fordítjuk, és 1-et hozzáadunk.

```
   11100100   (eredeti)
   00011011   (fordított)
 +        1
 ----------
 % 00011100   (kettes komplemens)
```

A kapott érték $1C, ami a decimális 28.

Még egyszer hangsúlyozzuk, hogy összeadási művelet előtt a C kapcsolót 0-ra, kivonás előtt 1-re kell állítani, jelezve ezzel, hogy nem volt átvitel.
Az utasításkódok:

Címzésmód	SBC
közvetlen	$E9
abszolút	$ED
nulláslap	$E5
abszolút, X-szel indexelt	$FD
abszolút, Y-nal indexelt	$F9
nulláslap, X-szel indexelt	$F5
indirekt indexelt	$F1
indexelt indirekt	$E1

3.5 A logikai utasítások

Logikai műveletek végrehajtása során az aritmetikai műveletekhez hasonlóan az egyik operandusnak az akkumulátorban, a másiknak egy megadott tárcímen kell lennie. Az eredmény ismét az akkumulátorba kerül. A 6510-es processzor három logikai műveletet ismer.

A logikai ÉS művelet

A két operandus között a processzor bitenkénti ÉS műveletet végez a következő művelettábla szerint:

0 AND 0=0
0 AND 1=0
1 ABD 0=0
1 AND 1=1

Hajtsuk végre az

AND #$37

műveletet, miközben az akkumulátor tartalma $5D.

```
$5D 01011101
$37 00110111
$15 00010101
```

Az eredmény a bináris %00010101, ill. hexadecimálisan $15.
A megfelelő BASIC parancs, AND: azaz A = A AND $37, vagyis A = $5D AND $37, decimálisan A=93 AND 55.
Az eredmény decimális értéke 21, hexadecimális értéke $15.
Az AND logikai művelet az N és a Z kapcsolók értékét módosíthatja. Ha az eredmény nulla, akkor a Z kapcsoló értéke 1, ha az eredmény nagyobb, mint $7F (127), akkor pedig az N kapcsoló értéke lesz 1.
Az alábbi táblázat tartalmazza a címzésmódoknak megfelelő utasításkódokat.

Címzésmód	AND
közvetlen	$29
abszolút	$2D
nulláslap	$25
abszolút, X-szel indexelt	$3D
abszolút, Y-nal indexelt	$39
nulláslap, X-szel indexelt	$35
indirekt indexelt	$31
indexelt indirekt	$21

A logikai VAGY művelet

A processzor a logikai VAGY műveletet is bitenként végzi. Ha az akkumulátor, ill. az adott tárcím sorrendben megfelelő bitjei közül legalább az egyik értéke 1, az eredmény 1 lesz, egyébként 0. Ezt a művelettípust „megengedő VAGY" műveletnek is nevezzük.

```
0  ORA  0=0
0  ORA  1=1
1  ORA  0=1
1  ORA  1=1
```

Az eredmény 1, ha az első operandus és/vagy a második operandus értéke 1. A művelet eredményétől függően ismét a Z, ill. az N kapcsoló értékei módosulhatnak.

```
ORA #$37
```

Ha az akkumulátor tartalma $5D, a műveletet a processzor az alábbiak szerint végzi el:

```
$5D  01011101
$37  00110111
--------------
$7F  01111111
```

Az eredmény binárisan %01111111, ill. hexadecimálisan $7F.
Az utasítás BASIC megfelelője OR:

```
A=A OR $37
```

azaz A=$5D OR $37, ill. A=93 OR 55.
A BASIC utasítás eredménye a decimális 127, ami azonos a hexadecimális $7F értékkel.
A címzésmódoknak megfelelő utasításkódok:

Címzésmód	ORA
közvetlen	$09
abszolút	$0D
nulláslap	$05
abszolút, X-szel indexelt	$1D
abszolút, Y-nal indexelt	$19
nulláslap, X-szel indexelt	$15
indirekt indexelt	$11
indexelt indirekt	$01

A kizáró VAGY művelet

A kizáró VAGY logikai művelet abban tér el a megengedő VAGY művelettől, hogy az eredmény csak akkor lesz 1, ha a két operandus megfelelő bitje közül pontosan az egyik értéke 1.
A műveletábla:

0 EOR 0=0
0 EOR 1=1
1 EOR 0=1
1 EOR 1=0

Másképpen megfogalmazva, a művelet eredménye csak akkor 1, ha a két bit eltérő. A kizáró VAGY művelet eredményétől függően ismét a Z és N kapcsolók értéke módosulhat.
Az EOR utasításnak nincs közvetlen BASIC megfelelője, de a BASIC logikai utasításaiból elő tudjuk állítani a következőképpen: A EOR B: = (A OR B) AND NOT (A AND B)
Végül nézzünk egy példát:

```
EOR #$37
```

Ha az akkumulátor tartalma ismét $5D, a művelet:

```
$5D 01011101
$37 00110111
    --------
$6A 01101010
```

Az eredmény binárisan %01101010, ill. hexadecimálisan $6A.

A címzésmódoknak megfelelő utasításkódok:

Címzésmód	EOR
közvetlen	$49
abszolút	$4D
nulláslap	$45
abszolút, X-szel indexelt	$5D
abszolút, Y-nal indexelt	$59
nulláslap, X-szel indexelt	$55
indirekt indexelt	$51
indexelt indirekt	$41

A BIT utasítás

A BIT utasítás a 65XX processzorok sajátossága. Ez az utasítás csak a kapcsolókat állítja, miközben a regiszterek tartalma nem változik. A processzor logikai ÉS műveletet végez az akkumulátor és az adott tárcím tartalma között. Ha az eredmény nulla, a Z kapcsoló értékét 1-re állítja, egyébként törli. Ugyanakkor a tárcím hatodik bitjét a V kapcsolóba, hetedik bitjét pedig az N kapcsolóba tölti. A BIT művelettel megvizsgálhatjuk egy tetszőleges tárcím hatodik és hetedik bitjének tartalmát anélkül, hogy a regiszterek tartalmát elrontanánk.
Nézzünk erre egy példát:

```
BIT $1234
```

Tegyük fel, hogy az akkumulátor tartalma $10, a $1234 tárcím tartalma pedig $43. Az ÉS művelet eredménye:

```
$10  00010000
$43  01000011
AND  00000000
```

Az eredmény nulla, tehát a Z kapcsoló értéke 1 lesz. A V kapcsolóba az operandus hatodik bitje kerül, amelynek értéke példánkban 1, az N kapcsoló értéke pedig 0 lesz.
Az eredmény:

Z = 1 ;   V = 1 ;   N = 0.

A BIT utasítást kétféle címzésmóddal használhatjuk:

Címzésmód	BIT
nulláslap	$24
abszolút	$2C

3.6 Az összehasonlító utasítások

Ezekkel az utasításokkal összehasonlíthatjuk az akkumulátor, ill. a regiszterek tartalmát úgy, hogy közben tartalmuk nem változik meg. Az összehasonlítás eredményétől függően a C, az N, ill. a Z kapcsoló tartalma módosulhat.
A művelet operandusa a processzor bármely munkaregisztere lehet.

A CMP utasítás

A CMP utasítás hatására a processzor a megadott tárcím tartalmát kivonja az akkumulátor tartalmából. Ha kivonásnál alulcsordulás lép fel, törli a C kapcsolót, egyébként 1-re állítja. Ha az eredmény nulla, azaz a a két operandus egyenlő, a Z értéke 1, egyébként 0 lesz. Végül ha az eredmény nagyobb, mint $7F (127), akkor az N kapcsoló értéke 1, egyébként 0 lesz.
Nézzünk egy példát:

```
CMP #$30
```

Legyen először az akkumulátorban $50. A $50 – $30 kivonás közben nem keletkezik átvitel, így a kapcsolók értékei:

C = 1 ;   Z = 0 ;   N = 0

Ezzel szemben, ha az akkumulátor tartalma $30, a kivonás eredménye nulla, tehát a kapcsolók értékei:

C = 1 ;   Z = 1 ;   N = 0

Végül ha az akkumulátorban $10 van, a $10 – $30 = $F0, tehát alulcsordulás következett be. A kapcsolók értékei a művelet elvégzése után:

C = 0 ;   Z = 0 ;   N = 1

Érdemes megjegyezni, hogy a számok közötti nagyságrendi reláció a kapcsolók értékét hogyan befolyásolja, ill. a kapcsolók értékeiből hogyan következtethetünk a nagyságrendi relációra:

A kapcsoló értéke	A reláció
C = 1	> = (nagyobb vagy egyenlő)
Z = 1	= (egyenlő)
C = 0	< (kisebb)

A „nagyobb" reláció eldöntéséhez két kapcsoló vizsgálatára van szükség. Ekkor ugyanis:

Z = 0 és C = 1

Az összehasonlító utasítások, amelyek csak a kapcsolók értékeit módosítják, alapul szolgálnak a következő fejezetekben tárgyalandó feltételes utasításokhoz.

A címzésmódoknak megfelelő utasításkódok:

Címzésmód	CMP
közvetlen	$C9
abszolút	$CD
nulláslap	$C5
abszolút, X-szel indexelt	$DD
abszolút, Y-nal indexelt	$D9
nulláslap, X-szel indexelt	$D5
indirekt indexelt	$D1
indexelt indirekt	$C1

A CPX utasítás

Hasonlóan működik, mint a CMP utasítás, azonban most nem az akkumulátor tartalmához hasonlítjuk a tárcím tartalmát, hanem az X regiszteréhez. Az X regiszter tartalma nem változik. Ez esetben csak az alábbi címzésmódok érvényesek:

Címzésmód	CPX
közvetlen	$E0
abszolút	$EC
nulláslap	$E4

A CPY utasítás

Ugyanaz vonatkozik erre az utasításra is, mint az előzőre, azzal a különbséggel, hogy most az Y regiszter tartalma vesz részt az összehasonlításban.

3.7 A feltételes elágazások

Az elágazó utasítások megszabják a gépi kódú utasítások végrehajtásának sorrendjét. Ha nincs elágazás, a processzor az utasításokat a megadás sorrendjében hajtja végre. Az elágazásokat döntések előzik meg, a döntéseket pedig általában a kapcsolók értékei vezérlik. A döntésekben négy kapcsoló vesz részt: a Z, az N, a C, ill. a V kapcsoló. Minden kapcsolóhoz két feltételes ugróutasítás tartozik; ugrás, ha a kapcsoló értéke 1, ill. ugrás, ha a kapcsoló értéke 0. Mivel azt is tudatnunk kell a processzorral, hogy melyik utasításra adjuk a vezérlést, ezeknek az utasításoknak tartalmazniuk kell még egy operandust: az ugrási címet, amely egy 16 bites érték. Így a feltételes ugrási utasítások három byte-ot foglalnának el: egy byte-ot az utasításkód, kettőt pedig az ugrási cím. Minthogy azonban az ugrások általában kis távolságra vonatkoznak, kidolgoztak egy olyan címzésmódot, ami feleslegessé teszi a két byte-os ugrási címeket. Ez az ún. *relatív címzés*. Ha az ugrási címet nem a tár kezdetéhez, hanem az éppen végrehajtott utasítás címéhez képest adjuk meg, akkor általában elegendő nyolc bit az ugrási cím tárolására. Általában az éppen végrehajtott utasítás címét a processzor az utasításszámlálóban egyébként is jegyzi. A nyolc biten összesen 256 különböző ugrási címet adhatunk meg. A programon belül azonban hátrafelé is ugorhatunk, amelyet megkülönböztetésül az előre ugrástól, negatív számmal fejezhetünk ki. Így a 0-tól 255-ig terjedő számtartományt két részre osztjuk. Ha a hetedik bit értéke 1, a számot negatívnak, egyébként pozitívnak tekintjük. Ha a hetedik bit értéke 1, a számot egy negatív szám kettes komplemenseként kezeljük.

%10000000	$80	–128
%10000001	$81	–127
...	...	....
%11111110	$FE	–2
%11111111	$FF	–1
%00000000	$00	0
%00000001	$01	1
%00000010	$02	2
...	...	....
%01111110	$7E	126
%01111111	$7F	127

Vizsgáljuk meg, hogy hogyan lehet kiszámítani a relatív ugrási címet az utasítások között lévő távolság alapján. A vonatkoztatási pont a feltételes ugrást követő utasítás címe. Tegyük fel, hogy az ugró utasítás a $C47A, az az utasítás pedig, ahová ugrani akarunk, a $C4BF címen van.

$C47A	az ugróutasítás címe
$C47C	a következő utasítás címe
$C4BF	a célutasítás címe

A relatív cím:

$C4BF – $C47C = $43

Ezt a címet kell megadni az ugróutasítás operandusaként.
Hogyan kell kiszámítani az ugrási címet visszafelé ugrásnál?

Tegyük fel, hogy a célutasítás a $C440-es címen van. A számítás egyik módozata azonos az előzővel, egyszerűen képezzük a két cím különbségét:

$C440 – $C47C = $FFC4 + alulcsordulás

Az ugróutasítás operandusát az eredmény alsó byte-ja adja meg.
A számítás másik módozata az, hogy a különbséget fordítva képezzük, majd vesszük az eredmény kettes komplemensét.

$C47C – $C440 = $3C

A kettes komplemens képzése:

```
  %00111100
  %11000011
+         1
  ---------
  %11000100 = $C4
```

Az eredmény mindkét számítási módszerrel ugyanaz.

A relatív címzésnek több előnye van. Egyrészt a takarékos tárkihasználás, hiszen így egy utasítás három byte helyett csak kettőt foglal el. Másrészt a processzor sokkal gyorsabban végre tudja hajtani a két byte-os utasításokat, mint a három byte-osokat. A relatív címzés legfontosabb előnyét azonban mégiscsak az jelenti, hogy a címeket a programon belüli elhelyezkedésük alapján viszonyítjuk egymáshoz. Ebből ugyanis az következik, hogy amikor a programot az egyik tárterületről áthelyezzük a másikra, nem kell az ugrási címeket megváltoztatnunk, hiszen a programon belüli távolságok nem változnak meg. Ha abszolút címekkel dolgoznánk, minden programáthelyezéskor át kellene írnunk a programban szereplő összes ugrási címet.
A relatív címzés hátránya ugyanakkor a viszonylag kis átfogható tartomány. Ezzel a módszerrel előre mindössze 129 byte, hátra pedig 126 byte távolságot tudunk megcímezni. Valójában azonban ritkán fordul elő, hogy ennél nagyobb címtartományt kezelő programot kell megírni.

Ne okozzon gondot az Olvasónak, ha a relatív címek kiszámítási módszereit nem sikerült tökéletesen megértenie. A későbbiekben ugyanis látni fogjuk, hogy a számítást elvégzi helyettünk az Assembler program, a programozónak csak az ugrási címet kell megadnia. Az Assembler program azt is megvizsgálja, hogy az ugrási cím beleesik-e a megengedett címtartományba.
Most vegyük sorra a tényleges elágazó utasításokat.

Elágazás a Z kapcsoló értéke alapján

A Z kapcsoló értékét vizsgálva „elágazik, ha egyenlő" a BEQ utasítás (Branch on EQual). Ha a Z kapcsoló értéke 0, az „elágazik, ha nem egyenlő", azaz a BNE (Branch on Not Equal) utasítással ugorhatunk a kívánt címre.

Elágazás a C kapcsoló alapján

A C kapcsoló értéke alapján „elágazik, ha az átvitel 1" a BCS (Branch on Carry Set) utasítás, amelynek párja az „elágazik, ha az átvitel 0", azaz a BCC (Branch on Carry Clear) utasítás.

Elágazás az N kapcsoló alapján

Az N kapcsoló értékétől függő két elágazó utasítás az „elágazás, ha negatív",.BMI (Branch on MInus), ill. az „elágazás, ha pozitív", BPL (Branch on PLus) utasítások.

Elágazás a V kapcsoló alapján

A V kapcsoló értékétől függő elágazások a BVS (Branch on oVerflow Set), ill. a BVC (Branch on oVerflow Clear). A BVS utasítás elágazást okoz, ha a V kapcsoló értéke 1, a BVS pedig, ha a V kapcsoló értéke 0 volt.

A feltételes utasítások kódjai:

Utasítás	Kód
BEQ	$F0
BNE	$D0
BCS	$B0
BCC	$90
BMI	$30
BPL	$10
BVS	$70
BVC	$50

3.8 Az ugróutasítások

Ellentétben a fenti feltételes ugróutasításokkal, az alábbi feltétel nélküli ugróutasítások abszolút címekkel dolgoznak. Végrehajtásuk nem függ semmilyen feltételtől.
A feltétel nélküli ugrásoknál használhatjuk az indirekt címzést, amelynél az operandus nem magát az ugrási címet, hanem azt a tárcímet adja, amely tartalmazza az ugrási címet. Az ugrási címet két egymást követő tárcím tartalma határozza meg (alsó byte, felső byte). A feltétel nélküli elágazás a JMP utasítás. Nézzünk erre egy példát:

JMP ($0302) indirekt ugrás a $0302 címre.

Az ugrási hely tényleges címét a $0302 és a $0303 cím tartalmazza. Ha ezeken a címeken $40 és $C8 áll, a tényleges ugrási cím: $C840.

Címzésmód	JMP
abszolút	$4C
indirekt	$6C

A Commodore 64-es a címzésmódok gazdag lehetőségét szolgáltatja. A $300–$330 tárcímeken találjuk a tárban az úgynevezett ugrási vektorokat. Ennek óriási előnye, hogy ha a saját igényeink szerint a rendszerben és a BASIC-ben változtatni akarunk, mindössze ezeket a vektorokat kell megváltoztatnunk.

3.9 A számláló utasítások

A ciklusutasítások hatékony végrehajtásához a 6510-es processzor tartalmaz olyan utasításokat, amelyek eggyel növelik (csökkentik) a regiszterek vagy egy adott tárcím tartalmát. A növelő utasítások az elágazásokkal együtt megfelelnek a BASIC-beli NEXT utasításnak. A csökkentő utasítás a STEP-1 BASIC utasításnak felel meg.

INX

Az INX utasítás az X regiszter tartalmát növeli eggyel, az eredménynek megfelelően az N, illetve a Z kapcsoló értéke módosul.

X = X+1

Ha a $FF értékét növelnénk, az átvitelt a gép figyelmen kívül hagyja, és a Z kapcsoló értéke 1 lesz.

INY

Az Y regiszter tartalmát növeli eggyel. A kapcsolók beállítása a fentiekhez hasonlóan történik. A 6510-es processzor nem tartalmaz olyan utasítást, amely az akkumulátor tartalmát növelné 1-gyel.

INC

Az adott tárcím tartalmát növeli eggyel. Az eredménytől függően ismét változik a Z és az N kapcsolók értéke. Az előbb ismertetett utasításoktól annyiban különbözik, hogy először beolvassa egy tárcím tartalmát, eggyel megnöveli, majd ezt az értéket ismét visszaírja a címre (READ – MODIFY – WRITE). Az eddig megismert utasítások vagy csak írtak, vagy csak olvastak egy tárcímről, egyszerre írást és olvasást azonban nem végeztek. Az akkumulátor tartalmát az INC utasítással sem tudjuk megváltoztatni.
BASIC-ben:

```
POKE M,PEEK(M)+1
```

ahol M a tárcím.

Az alábbiakban a fenti növelő utasítások csökkentő párjait ismertetjük.

DEX

Az X regiszter tartalmát eggyel csökkenti. Ha a regiszter tartalma $00-ról $FF-re csökken, a C kapcsoló értéke nem módosul. A Z és az N kapcsolók BASIC megfelelője:

X = X – 1

DEY

Ugyanaz, mint a DEX, az Y regiszterre vonatkozóan.

DEC

Egy megadott tárcím tartalma csökken eggyel, anélkül, hogy eközben az akkumulátor tartalma elveszne. Erre az utasításra értelemszerűen ugyanaz vonatkozik, mint az INC-re.

Az utasításkódok táblázata:

Utasítás	Kód
INX	$E8
INY	$C8
DEX	$CA
DEY	$88

Címzésmód	INC	DEC
abszolút	$EE	$CE
nulláslap	$E6	$C6
abszolút, X-szel indexelt	$FE	$DE
nulláslap, X-szel indexelt	$F6	$D6

3.10 A kapcsolók értékét módosító utasítások

A kapcsolók tartalmát közvetlenül a programból is megváltoztathatjuk. Ezek az utasítások egy byte hosszúak, ugyanis operandusra nincs szükségük.

A C (átviteli) kapcsoló értékét módosító utasítások:

SEC (set carry: C = 1)
CLC (clear carry: C = 0)

A SEC utasítást minden kivonás, a CLC utasítást minden összeadás előtt végre kell hajtani.

A D (decimális) kapcsoló értékének módosítása:

A D kapcsoló tartalma dönti el, hogy a műveletet (összeadást vagy kivonást) a processzor bináris (D = 0) vagy decimális (D = 1) aritmetikával végzi el. Ha D = 1, a processzor úgynevezett BCD számokkal (binárisan kódolt decimális) végzi a műveleteket.

SED : D = 1
CLD : D = 0

Az I (megszakítási) kapcsoló értékének módosítása:

Az I kapcsoló értéke dönti el, hogy a processzor egy megszakítást (interrupt) engedélyez vagy sem. Ha a SEI utasítással az I értékét 1-re állítjuk, a megszakítást a processzor letiltja, ha a CLI utasítással az I értékét 0-ra állítjuk, a processzor a megszakítást engedélyezi.

A V (túlcsordulási) kapcsoló értékének módosítása:

A V kapcsolót közvetlenül utasítással csak törölni tudjuk. Erre szolgál a CLV (CLear oVerflow) utasítás.

Az utasítások kódjai:

Utasítás	Kód
CLC	$18
SEC	$38
CLD	$D8
SED	$F8
CLI	$58
SEI	$78
CLV	$B8

3.11 Az eltolási utasítások

A 6510-es processzor tartalmaz egy olyan utasításcsoportot, amelynek nincs BASIC megfelelője. Ezek az utasítások valamely tárcím vagy az akkumulátor bináris tartalmát egy pozícióval jobbra vagy balra eltolják. Ha az utasítás az akkumulátor tartalmára vonatkozik, úgynevezett akkumulátor-címzésről beszélünk. A címzésmód szerint ezek az utasítások lehetnek egy, kettő vagy három byte hosszúak. Ha az eltolás egy tárcímre vonatkozik, akkor az INC és a DEC utasításokhoz hasonlóan a műveletet a processzor író, olvasó ciklusban végzi el, és az akkumulátor tartalmát változatlanul hagyja.

ASL

Az ASL utasítás jelentése: aritmetikai eltolás balra (Arithmetic Shift Left). Az utasítás hatására a megcímzett byte tartalma egy hellyel balra tolódik. Az eltolás során a nulladik bit értéke 0 lesz, a hetedik bit értéke pedig bekerül a C kapcsolóba. Nézzünk példaként egy olyan ASL utasítást, amely az akkumulátorra vonatkozik. Legyen az akkumulátor tartalma $47:

```
$47 %01000111
    %10001110    $8E, C = 0
```

A fenti példában az akkumulátor új tartalma $8E, a C kapcsoló értéke 0 lesz, mivel a hetedik biten eredetileg 0 volt. Az akkumulátor új tartalma az eredeti érték kétszerese. Ha egy decimális számot egy helyiértékkel balra tolunk, a kapott szám az eredeti tízszerese lesz. Bináris rendszerben az eltolás eredményeként kapott szám az eredeti szám kétszerese.
Nézzünk még egy példát. Legyen az akkumulátor tartalma $CD:

```
$CD %11001101
    %10011010    $9A, C = 1
```

Ez esetben is az eredeti szám dupláját kapjuk, ha figyelembe vesszük a C kapcsoló értékét. A $CD (205) kétszerese a $19A (410).

LSR

Az LSR utasítás jelentése: logikai eltolás jobbra (Logical Shift Right). Hatása ugyanaz, mint az ASL utasításé, azzal a különbséggel, hogy a tárcím tartalma most egy hellyel jobbra tolódik. A hetedik bit értéke 0 lesz, a nulladik bit eredeti értéke pedig átkerül a C kapcsolóba. Legyen az akkumulátor tartalma $CA, és nézzük meg mi a hatása a következő utasításnak:

```
LSR A
$CA  %11001010
     %01100101    $65, C = 0
```

Az eredmény az eredeti érték fele, a C kapcsoló értéke pedig 0, hiszen a nulladik biten eredetileg 0 volt. A C kapcsoló értéke az LSR művelet elvégzése után a kettővel való osztás maradékát adja. Az LSR művelettel eldönthetjük, hogy egy szám páros vagy páratlan, és a BCC vagy BCS utasításokkal elágaztathatjuk a programot az eredménytől függően. Ha az LSR utasítás egy tárcímre vonatkozik, az akkumulátor tartalma változatlan marad.

ROL

A ROL utasítás jelentése: ciklikus elforgatás balra (Rotate Left). Az elforgatás során a processzor kilenc bittel dolgozik, kiegészítve a regisztert a C kapcsolóval. A C kapcsoló értéke a nulladik bitre kerül, a hetedik bit eredeti értéke pedig bekerül a C kapcsolóba.
Tegyük fel, hogy az akkumulátor tartalma $4B, a C kapcsoló értéke pedig 1, és vizsgáljuk meg, hogy mi történik a következő utasítás hatására:

ROL A
$4B %01001011 C = 1
$97 %10010111 C = 0

Minden bit egy hellyel balra tolódott. A C kapcsoló értéke az eltolás során a felszabadult nulladik bitre, a túlcsordult hetedik bit értéke pedig a C kapcsolóba került.

ROR

A ROR utasítás: elforgatás jobbra (Rotate Right). A ROL utasítással szemben a ROR a regiszter tartalmát egy pozícióval ciklikusan jobbra tolja. A C kapcsoló tartalma most a felszabadult hetedik bitre, a túlcsordult nulladik bit értéke pedig a C kapcsolóba kerül. Tegyük fel, hogy az akkumulátor tartalma $89, a C kapcsoló értéke pedig 0. Ekkor ROR A:

$89 %10001001 C = 0
$44 %01000100 C = 1

A $89 hexadecimális számból $44-et kaptunk, a kettővel való osztás maradékát pedig ismét a C kapcsoló értéke mutatja.
Az eltolási utasítások elvégzésekor a C kapcsoló értékén kívül a Z és az N kapcsolók értéke is módosulhat, az eltolás eredményétől függően.

A címzésmódoknak megfelelő utasításkódok:

Címzésmód	ASL	LSR	ROL	ROR
akkumulátor	$0A	$4A	$2A	$6A
abszolút	$0E	$4E	$2E	$6E
nulláslap	$06	$46	$26	$66
abszolút, X-szel indexelt	$1E	$5E	$3E	$7E
nulláslap, X-szel indexelt	$16	$56	$36	$76

3.12 A szubrutin (alprogram) utasítások

A programok alprogramokra bontása nagyon fontos programozástechnikai módszer. A BASIC-ben a GOSUB utasítással ugorhatunk egy alprogramba, majd a RETURN utasítással térhetünk vissza a főprogramba.
Hogyan különböztetjük meg az egyszerű ugróutasítást a GOTO-t (JMP) egy alprogramhívástól? Alprogram hívásakor a BASIC interpreternek meg kell jegyeznie, honnan történt a hívás, mivel a RETURN után ezen a címen kell a program végrehajtását folytatnia. A BASIC ezt automatikusan megjegyzi, ennek ellenére nem árt, ha tudjuk, valójában mi történik.
A processzor egy alprogram hívásakor az utasításszámláló pillanatnyi értékét megőrzi a veremben. A verem a $0100 – $01FF (256–511) fix tárcímeken helyezkedik el. A veremmutató tartalma megadja, hogy a vermen belül hol kell keresni a visszatérési címet.
Mi történik tehát egy alprogram hívásakor?
A processzor veszi a pillanatnyi címet (+2) és elhelyezi egy alsó, illetve felső byte-ban. A felső byte helye a $100 + SP cím. Ekkor az SP (veremmutató) tartalma eggyel csökken, és az alsó

byte bekerül a verembe (a $100+SP címre). Végül a veremmutató értéke még eggyel csökken, és a program végrehajtása az ugrási címen folytatódik. Ha a processzor RETURN utasításhoz ér, a folyamat megfordítva zajlik le. A mutató tartalma eggyel nő, betöltődik egy byte a veremből (a $100+SP címről). Az utasításszámláló ezt a byte-ot alsó byte-ként kezeli. Ekkor a mutató értéke ismét eggyel nő, majd a veremből betöltődik a felső byte. Így az utasításszámláló ismét a hívás pontját követő utasításra mutat, és a program végrehajtása innen folytatódik.

Tekintsük át mégegyszer a verem működését. Valahányszor a verembe bekerül egy érték, a mutató értéke eggyel csökken, egy byte visszaolvasásakor eggyel nő. A veremmutató felülről lefelé nő ($1FF-től $100-ra).

Nézzünk erre a folyamatra egy példát:

```
$C480 JSR $2000     SP=$FA
                 $01FA=$C4    SP = SP - 1
                 $01F9=$82    SP = SP - 1
                    SP=$F8
```

A fenti példában a $2000-es címre ugrunk, ahol az egyszerűség kedvéért álljon egy RETURN utasítás.

```
$2000 RTS        SP=$F8
                 SP=SP+1    PCL=($01F9)=$82
                 SP=SP+1    PCH=($01FA)=$C4
                 SP=$FA
```

A programszámláló tartalma $C482. Ezt az értéket eggyel megnövelve, megkapjuk a $C480-as címen lévő utasítást (rutinhívás) követő utasítás címét ($C483)

A verem „az utoljára be–először ki" elven működik, vagyis mindig a verembe utoljára beírt értéket olvassuk vissza. Így tudjuk beilleszteni a rutin végrehajtását a program végrehajtásába. Ha egy rutinból meghívunk egy további rutint, az utoljára végrehajtott RETURN utasítás hatására az utoljára elhelyezett visszaugrási cím kerül elő a veremből, tehát először az előző rutinba, majd onnan a főprogramba térünk vissza. Ha valaki jól megértette a verem munkamódszerét, ezt a tárterületet használhatja átmeneti adatterületként.

A rutinhívó és a visszatérő utasítás kódja:

Utasítás	Kód
JSR	$20
RTS	$60

3.13 A veremutasítások

A veremutasításokkal az akkumulátor és az állapotregiszterek tartalmát átmenetileg a veremben megőrizzük, majd visszatöltjük. Eközben a veremmutató automatikusan írás után eggyel csökken, olvasás után eggyel nő.

PHA

A PHA utasítás (PusH Accu) végrehajtásakor az akkumulátor tartalma a verembe kerül, és a veremmutató értéke eggyel csökken.

PHP

A PHP utasítás (PusH Processzor status) végrehajtásakor az állapotregiszter tartalma bekerül a verembe, a veremmutató tartalma eggyel csökken, az állapotregiszter tartalma változatlan marad.

PLA

A PLA utasítás (PulL Accu) a PHA utasítás ellentéte. Végrehajtásakor a veremmutató értéke eggyel nő, és egy byte tartalma a veremből az akkumulátorba töltődik. Az értéknek megfelelően az N és Z kapcsoló értéke módosul.

PLP

A PLP utasítás (PulL Processzor status) a PHP utasítás ellentéte. A verem tartalma bekerül az állapotregiszterbe.

Az utasításkódok táblázata:

Utasítás	Kód
PHA	$48
PHP	$08
PLA	$68
PLP	$08

3.14 A megszakítási utasítások

A 6510-es processzor lehetővé teszi, hogy egy program futását kívülről megszakítsuk. Ehhez egy úgynevezett megszakítási sorra (IRQ) van szükség, amely a processzort aktivizálja. A megszakítás módja hasonló az alprogram hívásához. A processzor megszakítja az éppen futó program munkáját, és az utasításszámláló tartalmát a verembe helyezi. Ugyanúgy az állapotregiszter tartalmát is elhelyezi a verembe, azért, hogy a program futását a megfelelő állapotról tudja folytatni.
Most következik az elágazás arra a címre, melyről a $FFFE és $FFFF címek mutatnak. Ezeknek a címeknek a tartalmát a processzor új utasításszámlálóként használja. A BRK utasítás programon belül okoz megszakítást.

Ekkor ugyanúgy, mint külső megszakításkor, az utasításszámláló és az állapotregiszter tartalma a verembe kerül.
A megszakításból való visszatérésre szolgál az RTI (ReTurn from Interrupt). Ez az utasítás visszaolvassa a veremből a programszámláló és az állapotregiszter tartalmát, és a program a kapcsolók tartalmának megváltoztatása nélkül folytatni tudja munkáját.

Utasítás	Kód
BRK	$00
RTI	$40

Végül egy utasítás, amelynek a feladata furcsa módon az, hogy ne csináljon semmit. Ennek segítségével írhatunk úgynevezett késleltető ciklusokat, ha például a program közben várakozási időre van szükségünk.

Utasítás	Kód
NOP	$EA

4. A GÉPI KÓDÚ PROGRAMOK TÁROLÁSA

Miután minden gépi kódú utasítást megismertünk, vizsgáljuk meg, hogyan kell a gépi kódú programot megírni és a tárba betölteni.
Ahogyan azt a korábbiakban láttuk, a gépi kódú program nem egyéb, mint utasításkódok sorozata, a hozzá tartozó operandusokkal.
Első példaként írjunk egy jelet a Commodore 64-es képernyőjére, gépi kódú programmal
A megfelelő BASIC parancs:

```
0 REM **** P1. ****
1 :
2 :
10 POKE 1024,1:REM AZ 'A' BETU KEPERNYOKODJA
20 POKE 55296,7:REM A SARGA SZIN KODJA
```

Ha végrehajtjuk ezt a két parancsot, a képernyő bal felső sarkában megjelenik egy sárga A betű. Most nézzük meg, hogyan lehet ugyanezt gépi kódú programmal megoldani.
A POKE utasítás megfelelője az STA utasítás, amely az akkumulátor tartalmát egy megadott tárcímre írja. Ezért a kívánt értéket először az akkumulátorba kell tölteni.

```
LDA #1
STA 1024
```

Hasonlóan a szín kódját is a megfelelő címre kell töltenünk.

```
LDA #7
STA 55296
```

Ha ezeket az utasításokat elküldjük a RETURN billentyűvel, a ?SYNTAX ERROR hibaüzenetet kapjuk.
A Commodore 64-es ugyanis közvetlen üzemmódban csak a BASIC parancsnokat érti meg. A feladatot tehát más módon kell megoldanunk. Emlékezzünk arra, hogy a gépi kódú program nem egyéb, mint utasításkódok és címek sorozata a tárban.
Először meg kell tehát határoznunk az utasítások kódjait a függelék alapján.
A közvetlen címzésű LDA utasítás kódja: $A9. Ezt követi az 1 operandus, majd egy abszolút címzésű STA utasítás kódja: $8D, majd a két byte-os tárcím, mint operandus: az alsó és felső byte. 1024-nek megfelel a hexadecimális $0400; 55296-nak a $D800.
Programunk tehát:

```
0 REM **** P2. ****
1 :
2 :
100 OPEN4,4
110 FOR I=1 TO 4:READ A$:PRINT#4,A$:NEXT
120 DATA LDA #$01,STA $0400,LDA #$07,STA $D800
130 CLOSE 4
```

Kódolva:

```
$A9, $01, $8D, $00, $04, $A9, $07, $8D, $00, $D8
```

A fenti hexadecimális számokat ebben a sorrendben kell elhelyeznünk a tárban. Keressünk a tárban egy olyan területet, amelyet a BASIC nem használ. Egy ilyen terület: 49152 – 53247, azaz $C000 – $CFFF, bőven elég gépi kódú programok elhelyezésére, hiszen négy kbyte-nyi. A betöltést kezdjük a 49152-es címen, és mivel BASIC programot használunk a töltéshez, váltsuk át a fenti számokat decimálissá:

169, 1, 141, 0, 4, 169, 7, 141, 0, 216

```
0 REM **** P3. ****
1 :
2 :
100 FOR I=0 TO 9
110 READ A:POKE 49152+I,A
120 NEXT
130 DATA 169,   0,141,  0,   4,169,   7,141,   0,216
```

A BASIC programot lefuttatva a gépi kódú programunk a megfelelő tárterületre kerül. Most már könnyen végrehajtjuk ezt a rövid kis gépi kódú programot SYS paranccsal. A SYS után meg kell adnunk a kezdőcímet, ami esetünkben 49152.

Legyünk azonban óvatosak! Mi történik ugyanis, miután a processzor az STA $D800 parancsot végrehajtotta? Betölti a következő tárcím tartalmát, és utasításkódként értelmezi. Ha most ott nem megfelelő érték áll, a gép határozatlan állapotba kerül, melyet esetleg csak kikapcsolással tudunk megszüntetni, és így elvész az eddigi munkánk.

Gondoskodnunk kell arról, hogy miután a processzor a 4. utasítást végrehajtotta, visszaadja a vezérlést a BASIC-hez. Le kell tehát zárnunk a programrészt egy RTS paranccsal:

POKE 49152 + 10,96

Most ténylegesen elindíthatjuk a programot a SYS 49152 paranccsal! Abban a szempillantásban megjelenik a képernyő felső sarkában a sárga A betű, és a gép a READY üzenettel visszajelentkezik. Ha az Olvasó a gépi kódú program elkészítését kissé körülményesnek találta, nincs egyedül a véleményével. Célszerű lenne ezt az eljárást lerövidíteni.

A BASIC betöltőprogramot csak abban az esetben hagyhatjuk el, ha van egy olyan programunk, amely a gépi kódú (helyesebben assembler) utasításokat, mint pl. az LDA # 1 közvetlenül tudja értelmezni, és automatikusan elhelyezi a megfelelő tárterületen. Ezt a programot assembler programnak hívják, és segítségével elkerülhetjük a fáradtságos átváltásokat (hexadecimálisról decimálisra és megfordítva), és kikerülhetjük az utasítástáblázatban való keresgélést is. A könyvben bemutatjuk egy ilyen assembler program teljes BASIC listáját. Mielőtt azonban ezt a programot ismertetnénk, bemutatunk néhány olyan szolgáltató programot, amelyek szintén megkönnyítik a gépi kódú programozást.

Az ún. monitorprogramokkal közvetlenül írhatunk a processzor tárcímeire, illetve regisztereibe; kiolvashatjuk, ill. módosíthatjuk tartalmukat. A legtöbb monitorprogrammal a gépi kódú programok lemezes tárolását és betöltését is elvégezhetjük.

Ha van monitorprogramunk, a gépi kódú programunkat hexadecimálisan is begépelhetjük. A monitorprogramok gyakran tartalmaznak ún. disassembler programot. A disassembler értelmezi a tárbeli gépi kódú programot, és kiírja mnemonikus formában. Könyvünkben ismertetünk egy disassembler programot, amellyel az Olvasó kilistázhatja pl. a BASIC interpreter, ill. az operációs rendszer érdekesebb részeit.

Gépi kódú programok készítése – Az assembler program

Az előző fejezetekben megismerkedtünk a gépi kódú utasításokkal és láttuk, hogy minden utasításhoz tartozik egy hexadecimális kód, ill. egy utasításszó, az ún. *mnemonik*.
A processzor csak a hexadecimális kódok formájában megadott utasításokat tudja végrehajtani, ami nehézkessé teszi a programozást. Ahhoz, hogy a kódok helyett a sokkal könnyebben kezelhető utasításszavakkal programozhassunk, szükség van egy közbenső fordítóprogramra, az ún. assemblerre.
A következő fejezetben közöljük egy assembler program BASIC listáját, és ismertetjük a program kezelését.

Írjunk egy olyan BASIC programot, amely kijelzi képernyőre a C 64-es teljes karakterkészletét. Hogy könnyebben megértse az Olvasó, a programot egy 0-tól 255-ig futó ciklussal szervezhetjük meg:

```
0 REM **** P4. ****
1 :
2 :
100 X=0
110 A=X
120 POKE 1024+X,A:REM KEPERNYOKOD
130 A=1
140 POKE 55296+X,A:REM SZINKOD
150 X=X+1
160 IF X<>256 THEN 110
170 END
```

Futtatva a programot, a képernyőn megjelennek a C 64-es által használt karakterek. A program futása kb. 7 másodpercig tart.
Írjuk át soronként a BASIC programot gépi kódú programmá!

100 X = 0 ⇒ LDX #$0

Az X változót az X regiszterben, az A változót pedig az akkumulátorban tároljuk.

110 A = X ⇒ TXA

Az X regiszter tartalmát áttöltjük az akkumulátorba, az X tartalma változatlan marad.

120 POKE 1024+X ⇒ STA 1024,X

Az akkumulátor tartalmát az 1024+X tárcímre visszük, indexelt címzéssel.

130 A = 1 ⇒ LDA #1

Az akkumulátorba 1-et (a fehér szín kódja) töltünk,

140 POKE 55296+X,A ⇒ STA 55196,X

és ezt az 55296+X tárcímre visszük.

150 X = X+1 ⇒ INX

Az X regiszter tartalmát 1-gyel megnöveljük.

160 IF X< >256 THEN 110 ⇒ ?

Ezt az utasítást kicsit át kell gondolnunk.
Ha az X regiszter tartalma még nem egyenlő 256-tal, vissza kell ugranunk a 110-es sorra, folytatni a karakterek kijelzését. Mi történik, ha az X regiszterben már 255 van, és ismét végrehajtjuk az INX utasítást (BASIC-ben a 150-es sort)? A 255-ből 256, ill. hexadecimálisan a $FF-ből $100 lesz, azaz ha az átvitelt nem vesszük figyelembe, az X regiszter tartalma $00. Honnan ismerjük fel a gépi kódú programban ezt a helyzetet? Valahányszor az X regiszter tartalma változik, az N és Z kapcsolók értéke is módosulhat. Ha a regiszter tartalma 0, a Z kapcsoló értéke 1 lesz, egyébként 0. Az elágazásról a Z kapcsoló értéke alapján dönthetünk. Mindaddig, míg a Z kapcsoló értéke 0, az X regiszter tartalma nem érte el a 256-ot, tehát a 11-es sorra kell visszaugranunk:

160 IF X< >256 THEN 110 ⇒ BNE ugrás a 110-re

A gépi kódú programban a BNE utasítás után meg kell adnunk azt a tárcímet, ahol a 110-es sornak megfelelő gépi kódú utasítás áll. Ezt a tárcímet még nem ismerjük. Kiszámításához tudnunk kell, hogy melyik tárcímen kezdődik a gépi kódú program, és milyen hosszúak (hány byte-ot foglalnak el) az egyes gépi kódú utasítások. Helyezzük el a programot a 49152-es ($C000-s) tárcímtől kezdve.

```
0 REM **** P5. ****
1 :
2 :
100 $C000   LDX #0
110 $C002   TXA
120 $C003   STA $0400,X
130 $C006   LDA #1
140 $C008   STA $D800,X
150 $C00B   INX
160 $C00C   BNE $C002
170 $C00E   RTS
```

Az utasításszámlálót aszerint léptettük, hogy az egyes utasítások hány byte-ot foglalnak el a tárban. A kapott címek alapján a 160-as sorból a $C002-es címre kell visszaugrani.
Írjuk át az utasításszavakat hexadecimális kódokká:

```
0 REM **** P6. ****
1 :
2 :
100 $C000   A2 00       LDX #0
110 $C002   8A          TXA
120 $C003   9D 00 04    STA $0400,X
130 $C006   A9 01       LDA #1
140 $C008   9D 00 D8    STA $D800,X
150 $C00B   E8          INX
160 $C00C   D0 ??       BNE $C002
170 $C00E   60          RTS
```

A 16-os sorban a BNE utasítás paramétere abszolút cím, ezt át kell számolnunk a programszámláló aktuális értékéhez viszonyított relatív címmé.
Képezzük a címek közötti pozitív különbség kettes komplemensét:

```
 $C00E
-$C002
------
$ 000C

$0C = %00001100
      %11110011
    +         1
      ---------
      %11110100 = $F4
```

A relatív cím (offset) $F4. A programot hexadecimálissá kódolt formában betölthetjük a tárba egy BASIC programmal, vagy a könyvben közölt *Egylépéses szimulátor* program M parancsával.
Betöltés után gépeljük be a

SYS 49152

parancsot. Egy szempillantás alatt megjelenik a képernyőn a teljes karakterkészlet. Az a program, amely BASIC nyelven több mint hét másodpercig futna, gépi kódú nyelven a másodperc törtrésze alatt befejezi a karakterek kiírását.
A gépi kódú program végrehajtási sebességét feltehetően sikerült érzékeltetnünk. A fantasztikus sebesség azonban nem vigasz a programozó számára, ha a gépi kódú programot a fenti, igen lassú és fáradtságos munkával kell megírnia.
A következő fejezetben ismertetjük az Assembler programot, amely a gépi kódú programok elkészítését gyorssá és kényelmessé teszi.

5. Az Assembler

Az Assembler* program felhasználása lehetővé teszi, hogy a gépi kódú programot éppúgy megszerkeszthessük, mintha BASIC program lenne. Megváltoztathatjuk a sorokat, új sort illeszthetünk be, a felesleges sorokat törölhetjük.

A programsor, a BASIC sorhoz hasonlóan, sorszámmal kezdődik. A sorszám után egy címke áll, ezt követi az assembler utasítás, majd a lehetséges operandusok. Az utasításoktól pontosvesszővel elválasztva megjegyzéseket is írhatunk, amit az Assembler program fordítás közben figyelmen kívül hagy. A pontosvessző megfelel a BASIC-beli REM utasításnak. Egy teljes assembler sor pl. a következő:

```
100  SZOVEG LDA $70,X ; KEZDOERTEK BETOLTESE
```

A megszerkesztett, ún. forrásprogramot (source program) lemezen tároljuk. Megkülönböztetésül a többi programoktól, az assembler forrásprogram mögé az SRC jelzést kell tennünk. Most betölthetjük az Assembler programot. Indítás (RUN) után az Assembler megkérdezi a lefordítandó program nevét, majd az adott néven tárolt forrásprogramot beolvassa a lemezről, és hexadecimális kódok formájában elhelyezi a tárban. Kívánságra olyan listát készít a forrásprogramról, amelyben a sorszámok és utasításszavak mellett a gépi kódok is megjelennek. A programlistát kérhetjük képernyőre vagy nyomtatóra.

A programozó a tényleges ugrási címek helyett szimbolikus címekkel dolgozhat. Az Assembler fordítás közben kiszámítja az ugrási címeket.

Nézzünk erre egy példát:

```
0 REM **** P7. ****
1 :
2 :
100         LDX #0
110 CIMKE   TXA
120         STA $0400,X
130         LDA #1
140         STA $D800,X
150         INX
160         BNE CIMKE
170         RTS
```

Programíráskor szimbolikus névvel jelöljük meg azt az utasítást, amelyre a későbbiek során ugrani akarunk. Ha az Assembler fordítás közben szimbolikus címet talál, tárolja a szöveget és a programszámláló aktuális értékét. A fenti példában a CÍMKE szimbolikus címhez a programszámláló $C002 értéke tartozik.

A 160-as sorban hivatkozunk a CIMKE szimbolikus címre. Az Assembler tudja, hogy a CIMKE a $C002-es tárcímű utasítás előtt áll. A tárolt címből és a programszámláló aktuális értékéből meghatározza az elágazó utasítás paraméterét, a relatív ugrási címet.

Fordítás közben az Assembler folyamatosan elhelyezi a tárban a talált utasítások gépi kódjait.

Mire az Assembler lefut, a tárban kész a futtatható gépi kódú program.

Hogyan állapítja meg az Assembler az ugrási címet akkor, ha a szimbolikus címre előbb hivatkozunk, mint definiáljuk? Például:

* A továbbiakban Assembler néven a könyvben ismertetett programra hivatkozunk.

```
0 REM **** P8. ****
1 :
2 :
100        LDA $40
110        BEQ TOVABB
120        LDX #$FF
130 TOVABB STX $D840
140        RTS
```

A 110-es sorban hivatkozunk a TOVABB címkére, holott az a 130-as sor előtt áll. Ahhoz, hogy minden ugrási címet egyértelműen kiszámíthasson, az Assembler fordítás előtt végigfutja az egész programot, és tárolja az összes címet. Általában minden assembler fordítóprogram két menetben dolgozik. Az első menetben (PASS 1) megkeresi és tárolja a szimbolikus címeket, a második menetben (PASS 2) pedig ténylegesen lefordítja a forrásprogramot.
Az assembler forrásprogram szerkesztése a BASIC interpreter alatt történik. Amikor BASIC programot írunk, és egy sor begépelése után megnyomjuk a RETURN billentyűt, a BASIC interpreter megvizsgálja a begépelt szöveget és az utasításszavakat 1 byte-os utasításkódok (tokenek) formájában elhelyezi a tárban. Ha az assembler program tartalmaz BASIC szavakat, és az interpreter ezeket kódolja, az Assembler fordító nem tudja a tárolt szöveget értelmezni. Hogy ez ne fordulhasson elő, az assembler program készítése előtt töltsük be és futtassuk le az alábbi BASIC programot.
Az assembler program tárolása után az eredeti állapotot a

SYS 53181

utasítással állíthatjuk vissza.

```
0 REM **** P9. ****
1 :
2 :
100 FOR I=53100 TO 53191
110 READ X : POKE I,X : S=S+X : NEXT
120 DATA 169,119,160,207,141,  2,  3,140,  3,  3, 96, 32
130 DATA  96,165,134,122,132,123, 32,115,  0,170,240,243
140 DATA 162,255,134, 58,144,  6, 32,121,165, 76,225,167
150 DATA  32,107,169,160,  0,162,  0,189,  0,  2,232,201
160 DATA  32,240,248,201, 48,144,  4,201, 58,144,240,153
170 DATA   0,  2,201,  0,240,  7,189,  0,  2,200,232,208
180 DATA 242,200,200,200,200,200, 76,162,164,169,131,160
190 DATA 164,141,  2,  3,140,  3,  3, 96
200 IF S <> 11096 THEN PRINT"HIBA A DATASORBAN..!!" : END
210 SYS 53100 : PRINT"OK..!!"
```

A fenti BASIC programot minden olyan lemezen célszerű tárolni, amelyre assembler forrásprogramokat mentünk. Ha az assembler programok szerkesztése előtt elfeledkezünk arról, hogy ezt lefuttassuk, nem kell újra begépelni az assembler programot. Ekkor ugyanis le kell futtatni a BASIC programot, be kell tölteni a lemezről az elrontott assembler forrásprogramot, majd minden sor után meg kell nyomni a RETURN-t. A forrásprogram kijavított változatát ismét tároljuk a lemezen.
Gyakorlásként gépeljük be a P7. mintaprogramot, egészítsük ki a 180 .EN programsorral, majd tároljuk a lemezen TESZT SRC néven.
A .EN a forrásprogram végét jelzi a fordítóprogramnak. Ne feledkezzünk meg a BASIC program lefuttatásáról! Töltsük be az Assembler programot, és indítsuk el. A képernyőn a következő sorok jelennek meg:

```
A FORRASPROGRAM NEVE?       TESZT
LISTA (I/N)                 ?  I
NYOMTATORA (I/N)            ?  N
```

Rövid idő elteltével megjelenik a képernyőn a PASS 1 felirat, és a lemezegységen kigyullad a piros lámpa.
Közben megjelennek a képernyőn az utasítások sorszámai 100-tól 180-ig, végül a PASS 2 felirat és a következő lista:

```
0 REM **** P10. ****
1 :
2 :
C000   A2 00      100          LDX    #0
C002   8A         110 MARKE    TXA
C003   9D 00 04   120          STA    $0400,X
C006   A9 01      130          LDA    #1
C008   9D 00 D8   140          STA    $D800,X
C00B   E8         150          INX
C00C   D0 F4      160          BNE    MARKE
C00E   60         170          RTS
                  180          .EN
```

A listázás befejeztével az Assembler megkérdezi, hogy tárolja-e lemezen a kapott gépi kódú programot.

```
TAROLJAM (I/N)   ? I
```

Ha igennel válaszolunk, a gépi kódú program TESZT.OBJ néven lemezre kerül. Az OBJ az objectprogram (tárgyprogram) angol szó rövidítése. Végül megjelenik a képernyőn egy statisztika a gépi kódú programról, ill. az esetleges hibaüzenet(ek).

```
C000/C00F/000F
A FORRASPROGRAM:TESZT.SRC
0 HIBA
```

Az Assembler utolsó szolgáltatása a szimbólumok táblázata:

```
SZIMBOLUMTABLAZAT I/N ? I
RENDEZVE          I/N ? N
CIMKE C002
```

A szimbólumokat kérhetjük abc-be rendezve is.
Ezzel a forrásprogram lefordítását (assemblálását) befejeztük. A TESZT.OBJ a forrásprogram közvetlenül futtatható gépi kódú változata. Győződjünk meg erről a

```
SYS 49152
```

parancs begépelésével. A képernyőn megjelenik a teljes karakterkészlet.
Foglaljuk össze az Assembler program működésére vonatkozó ismereteket.
A forrásprogram minden sora egy sorszámmal kezdődik, amelyet egy szimbólum(címke)

követhet. A szimbólum mögött áll az utasításszó, pl. LDA, majd az esetleges operandus(ok). Végül pontosvesszővel elválasztva az eddigiektől, a sort tetszőleges megjegyzésekkel zárhatjuk. Feltétlenül javasoljuk, hogy a kezdő programozó sok emlékeztető megjegyzést tegyen a programjába!

A szimbólumok maximum öt betűből állhatnak, és természetesen egy programon belül nem szabad két azonos szimbólumot használni. Célszerű szimbólumként emlékeztető szöveget írni! Sokkal könnyebben érthetőek azok az assembler programok, amelyben a címkék utalnak az utasítás céljára.

Például:

```
0 REM **** P11. ****
1 :
2 :
0400              70 VIDEO  =    $400
D800              80 SZIN   =    $D800
C000              90        *=   $C000
C000  A2 00      100        LDX  #0
C002  8A         110 MARKE  TXA
C003  9D 00 04   120        STA  VIDEO,X
C006  A9 01      130        LDA  #1
C008  9D 00 D8   140        STA  SZIN,X
C00B  E8         150        INX
C00C  D0 F4      160        BNE  MARKE
C00E  60         170        RTS
                 180        .EN
```

A fenti program rendezett szimbólumtáblázata:

CIMKE C002 SZIN D800
VIDEO 0400

A 90-es sorban egy új utasítást írtunk. Hatására az utasításszámláló felveszi a $C000 értéket. Ezt az utasítást minden program elejére elhelyezhetjük, így biztosítva, hogy a program az általunk megadott tárterületre kerüljön. Egy további szabad terület a tárban a kazetta puffer $33F-től $3FB-ig (828–1019). Ezt a területet a BASIC nem használja, ha az Assembler már előtte lefoglalta.

Mi az előnye a szimbolikus írásmódnak? Egyrészt az, hogy a nevekkel utalhatunk a változók tartalmára, mint a fenti példában (SZIN). Másrészt a szimbolikus elemekkel készült programot sokkal könnyebb átírni, megváltoztatni. Ha pl. a VIDEORAM-ot a megszokott helyről áthelyezzük, elég ha a VIDEO szimbólum értékadó utasítását kicseréljük.

Természetesen minél többször előfordul egy-egy szimbólum, annál célszerűbb fix értékek helyett szimbólumokkal dolgozni. Így az esetleges program-módosításkor csak a program elejét kell megváltoztatni.

Bemutatunk egy további, úgynevezett pszeudoutasítást vagy más néven assemblerdirektívát. Ezzel az utasítással egy kívánt számértéket vagy szöveget elhelyezhetünk a gépi kódú program belsejébe.

Az utasítás:

.BY

Az utasítás operandusa egy 0–255 közé eső szám, vagy egy változó, amelynek értéke 255 alatt van.

```
.BY 100
.BY $7F
.BY CR
```

Az utasításnak van egy további opciója is. Gyakran van szükségünk arra, hogy egy 16 bites számot két 8 bites részre, alsó, ill. felső byte-ra bontsunk. A felső byte-ot adja a „ > ", az alsó byte-ot pedig a „ < " jel.
Például:

```
0 REM **** P12/1. ****
1 :
2 :
100 CONST   =   $AB3F
110 .BY   <CONST
120 .BY   >CONST
```

A fenti utasításokkal a $3F és a $AB értékeket egymást követően elhelyeztük a programban. Ugyanez közvetlen címzéssel:

```
0 REM **** P12/2. ****
1 :
2 :
130 LDA   #<CONST
140 LDY   #>CONST
```

A nulláslap címzést az operandus elé beírt csillag (*) karakterről ismeri fel az Assembler. Olyan indexelt címzésnél, amely csak nullás lapcímekkel dolgozik, erre nincs szükség.
Például:

```
0 REM **** P13. ****
1 :
2 :
00B0                100 START   =     $B0
C000   AD B0 00     110         LDA   START
C003   AD B0 00     120         LDY   *START
C005   8D 27 00     130         STA   $27
C008   84 60        140         STY   *$60
C00A   24 B0        150         BIT   *START
```

Az utasítások formája meghatározza a végrehajtás módját. Ahol az operandus előtt csillag van, ott nulláslap címzéssel történik a végrehajtás, az utasítás két byte hosszú lesz (120-as, 140-es, 150-es sor).
Miután az Assembler program minden funkcióját megismertük, nincs más hátra, mint a gyakorlás. Az Assembler program listája:

```
0 REM **** P14. ****
1 :
2 :
100 REM 6510 ASSEMBLER  LE 12/83
110 PRINT CHR$(147):PRINT:PRINT:PRINT,"6510 - ASSEMBLER":PRINT:DG=8
120 INPUT "FORRAS-FILE-NEVE ";SN$
130 IF RIGHT$(SN$,4)=".SRC" THEN SN$=LEFT$(SN$,LEN(SN$)-4)
140 DD$="0":REM EGYSEGSZAM
150 INPUT "LISTA     I/N      ";A$:IF A$<>"I" THEN PM=1:GOTO 190
160 PF=4:PG=3
170 INPUT "NYOMTATORA I/N    ";A$:IF A$="I" THEN PG=4
```

```
180 OPEN PF,PG
190 GOSUB 5000:REM TABLAZAT FELEPITES
200 A=0:AD=49152:PRINT:PRINT:PA=A
210 PRINT "PASS 1":GOSUB 4000:PRINT "PASS 2":FF%=0:FE%=0
220 OP$=DD$+":"+SN$+".SRC"
230 OPEN 8,DG,0,OP$
240 GET#8,A$,A$:REM KEZDOCIM
250 IF PM=1 THEN PRINT CHR$(145),,ZN$
260 F%=0:IF AD>65535 THEN PRINT:PRINT:PRINT"TAR-TULLEPES!":GOTO 1000
270 A=AD:GOSUB 3240:PR$=A$+" ":GOSUB 2000:IF LEFT$(X$,3)=".EN" THEN 1000
280 XX$=LEFT$(X$,1):IF XX$="*" THEN PR$="                        ":LN$="    "
290 IF XX$="." OR XX$="*" ORXX$="=" THEN GOSUB 2900:GOTO 380
295 IF XX$=""THEN PR$=PR$+"           ":GOTO 430
300 ON LM% GOTO 320
310 SA=OF+AD:PA=AD:LM%=1
320 XX$=LEFT$(X$,3):FOR J=0 TO NN%:IF XX$=MN$(J) THEN 350
330 NEXT
340 FL$(1)="A":A%=1:F%=1:GOSUB 1520:GOTO 370
350 GOSUB 2400:F%=0:IFT%=5 AND T%(J,9)>0 THEN T%=9:REM RELATIV
360 ON T%+1 GOSUB 500,600,600,600,600,800,800,800,500,900,600,600,800
370 POKE OF+AD,A
380 AD=AD+A%:IF LEFT$(X$,2)="*=" THEN 400
390 LX=AD
400 REM        ************  KIIRATAS
410 IF F%=0 THEN IF FL$(0)=" " AND FL$(1)=" " AND FL$(2)=" " THEN 430
420 BS%=BS%+1
430 ON PM GOTO 250
440 Y$=LEFT$(Y$+"           ",11):FOR I=1 TO 3:PRINT#PF,FL$(I);:NEXT
450 PRINT#PF,PR$ ZN$ LN$ "   " LEFT$(X$+"      ",6) Y$ " " RM$
460 GOTO 250
500 REM EGY BYTE-OS UTASITASOK
510 A%=1:A=T%(J,T%):IF A<0 THEN FL$(2)="A":GOTO 1510
520 GOSUB 3240:PR$=PR$+RIGHT$(A$,2)+"         ":RETURN
600 REM KET-BYTE-OS UTASITASOK
610 A=T%(J,T%):IF A<0 THEN FL$(2)="A":GOTO 1500
620 GOSUB 3240:PR$=PR$+RIGHT$(A$,2)
630 YY$=YA$:IF LEFT$(YY$,1)="#" THEN YY$=MID$(YY$,2)
640 IF LEFT$(YY$,1)="*" THEN YY$=MID$(YY$,2)
650 A%=2:IF LEFT$(YY$,1)=">" OR LEFT$(YY$,1)="<" THEN : YY$=MID$(YY$,2)
660 A$=LEFT$(YY$,1):IF A$="$" OR A$>"/" AND A$<":" THEN A$=YY$:GOTO 690
670 SL$=YY$:GOSUB 4500
680 A$="$"+HE$
690 GOSUB 3100
700 IF LEFT$(YA$,2)="#>" THEN A=INT(A/HI)
710 IF LEFT$(YA$,2)="#<" THEN A=A-INT(A/HI)*HI
720 IF A>LO THEN FL$(2)="0":F%=1:A=0
730 GOSUB 3240:POKE OF+AD+1,AL%:PR$=PR$+" "+RIGHT$("00"+A$,2)+"      "
740 A=T%(J,T%):RETURN
800 REM HAROM BYTE-OS UTASITASOK
810 A%=3
820 A=T%(J,T%)
830 GOSUB 3240:PR$=PR$+RIGHT$(A$,2)
840 A$=LEFT$(YA$,1):IF A$="$" OR A$>"/" AND A$<":" THEN A$=YA$:GOTO 870
850 SL$=YA$:GOSUB 4500
860 A$="$"+HE$
870 GOSUB 3100:GOSUB 3240:PR$=PR$+" "+RIGHT$("00"+A$,2)+" "+LEFT$(A$,2)+"   "
880 POKE OF+AD+1,AL%:POKE OF+AD+2,AH%
890 A=T%(J,T%):RETURN
900 REM RELATIV
910 A%=2
920 A=T%(J,T%):GOSUB 3240:PR$=PR$+RIGHT$(A$,2)
930 A$=LEFT$(Y$,1):IF A$="$" OR A$>"/" AND A$<":" THEN A$=Y$:GOTO 960
940 SL$=Y$:GOSUB 4500
950 A$="$"+HE$
960 GOSUB 3100:IF FL$(2)="U" THEN A=AD+2
970 DF=A-(AD+2):IF DF<-128 OR DF>127 THEN FL$(3)="R":F%=1:DF=0
980 A=DF AND LO:GOSUB 3240
```

```
990 PR$=PR$+" "+RIGHT$(A$,2)+"      ":POKE OF+AD+1,A:A=T%(J,T%):RETURN
1000 PR$="                    ":IF F%=0 THEN 1020
1010 BS%=BS%+1
1020 IF AE<AD+OF THEN AE=AD+OF
1030 ON PM GOTO 1060
1040 FOR I=0 TO 3:PRINT#PF,FL$(I);:NEXT
1050 PRINT#PF,PR$ ZN$ LN$ "  " LEFT$(X$+"     ",6) Y$ " " RM$
1060 CLOSE 8:INPUT "LEMEZRE ?    (I/N)     ";A$:IF A$<>"I" THEN 1130
1070 A$=DD$+":"+SN$+".OBJ":POKE 186,DG
1080 A%=LEN(A$):POKE 183,A%:POKE 187,681 AND LO:POKE 188,681/HI
1090 FOR I=1 TO A%:POKE 680+I,ASC(MID$(A$,I)):NEXT:REM FILENAME
1100 A=SA:GOSUB 3240:POKE 251,AL%:POKE 252,AH%:REM KEZDOCIM
1110 A=AE:GOSUB 3240:POKE 781,AL%:POKE 782,AH%:REM VEGCIM
1120 POKE 780,251:SYS 65496:REM SAVE
1130 A=PA:GOSUB 3240:PA$=A$:A=AD:GOSUB 3240:AD$=A$:A=AD-PA:GOSUB 3240
1140 BA$=A$:ON PM GOTO 1180
1150 PRINT#PF:PRINT#PF,PA$" / "AD$" / "BA$
1160 PRINT#PF,"A FORRAS FILE: "SN$+".SRC"
1170 PRINT#PF,BS%" HIBA  ":PRINT#PF:CLOSE8:CLOSE15
1180 INPUT "SZIMBOLUM TABLA I/N ";Z$:IF Z$<>"I" THEN 1400
1190 MX=2:IF PG>3 THEN PRINT#PF,CHR$(12):MX=5
1200 INPUT "RENDEZES  I/N ";Z$:IF Z$="I" THEN 1306
1210 REM
1220 M%=0:P$="":FOR I=LL% TO UL%
1230 IF LB$(I)="      " THEN 1290
1240 P$=P$+LB$(I)+"  "+HE$(I)+"      ":M%=M%+1
1250 IF M%<>MX THEN 1290
1260 ON PM GOTO 1280
1270 PRINT#PF,P$
1280 P$="":M%=0:IF I>=UL% THEN 1400
1290 NEXT I:IF P$<>"" THEN 1260
1300 HI$=CHR$(127)+CHR$(127)+CHR$(127)+CHR$(127)+CHR$(127):F%=0:REM RENDEZES
1310 M%=0:SL$=HI$:FOR I=LL% TO UL%:IF LB$(I)="      " THEN 1340
1320 IF LB$(I)<SL$ THEN SL$=LB$(I):M%=I+1
1330 UL%=I
1340 NEXT I:IF F%<MX THEN 1360
1350 F%=0:IF PM=0 THEN PRINT#PF
1360 IF M%=0 THEN 1400
1370 ON PM GOTO 1390
1380 PRINT#PF,SL$"  "HE$(M%-1)"   ";
1390 LB$(M%-1)="      ":F%=F%+1:GOTO 1310
1400 REM
1410 IF PG=4 THEN PRINT#PF,CHR$(12)
1420 CLOSE PF:END
1500 POKE OF+AD+2,0:REM NOP-FUELLER
1510 POKE OF+AD+1,0
1520 A=0:PR$=PR$+NP$(A%):RETURN
1600 IF LEFT$(LN$,1)="." THEN I=-1:RETURN
1610 IF MID$(LN$,4,1)<>" " THEN I=NN%+1:RETURN
1620 MN$=LEFT$(LN$,3):REM LABEL=MNEMONIC ?
1630 FOR I=0 TO NN%:IF MN$<>MN$(I) THEN NEXT
1640 RETURN
2000 GET#8,A$,B$:IF A$+B$="" THEN 2290:REM
2010 GET#8,Z1$,Z2$
2020 ZN=ASC(Z1$+CHR$(0))+HI*ASC(Z2$+CHR$(0))
2030 ZN$=RIGHT$("    "+STR$(ZN),5)+" "
2040 GOSUB 2300:IF FF%THEN RETURN
2050 LN$="":X$="":Y$="":RM$="":X%=0
2060 FOR I=0 TO 3:FL$(I)=" ":NEXT I:IF Z$="*" THEN 2190
2070 IF Z$=";" THEN 2280
2080 REM CIMKENEV
2090 IF Z$=" " OR FF% THEN LN$=LEFT$(LN$+"     ",5):GOTO 2120
2100 LN$=LN$+Z$:IF LEN(LN$)=6 THEN X%=1:FL$(0)="L"
2110 GOSUB 2300:GOTO 2090
2120 GOSUB 1600:IF I<=NN% THEN X$=LN$:LN$="     ":GOTO 2200
2130 X%=ASC(LN$):IF X%<65 OR X%>90 THEN FL$(0)="S"
2140 REM OPERATION
```

```
2150 GOSUB 2300:IF FF% THEN RETURN
2160 IF Z$<>" " THEN 2190
2170 GOTO 2150
2180 GOSUB 2300:IF FF% THEN RETURN
2190 IF Z$<>" " THEN X$=X$+Z$:GOTO 2180
2200 IF FF% THEN RETURN
2210 IF Z$=";" THEN 2280
2220 IF Z$<>" " THEN 2260:REM OPERANDUS
2230 GOSUB 2300:IF FF% THEN RETURN
2240 GOTO 2200
2250 GOSUB 2300:IF FF% THEN RETURN
2260 IF Z$<>" " THEN Y$=Y$+Z$:GOTO 2250
2270 GOSUB 2300:IF FF% THEN RETURN:REM MEGJEGYZES
2280 RM$=RM$+Z$:GOTO 2270
2290 X$=".EN":RM$="END FELTETELEZVE":LN$="        ":Y$="":ZN$="          ":RETURN
2300 GET#8,Z$:FF%=Z$="":RETURN
2400 REM CIMZESMOD
2410 IF Y$="" THEN T%=8:RETURN:REM IMPLICIT
2420 YA$=Y$:IF LEFT$(YA$,1)="(" THEN YA$=MID$(YA$,2)
2430 IF RIGHT$(YA$,1)=")" THEN YA$=LEFT$(YA$,LEN(YA$)-1)
2440 IF RIGHT$(YA$,3)="),Y" THEN YA$=LEFT$(YA$,LEN(YA$)-3)
2450 IF RIGHT$(YA$,2)=",Y" OR RIGHT$(YA$,2)=",X" THEN YA$=LEFT$(YA$,LEN(YA$)-2)
2460 Z$=Y$:K$=LEFT$(Y$,1)
2470 IF Z$="A" THEN T%=0:RETURN:REM AKKUMULATOR
2480 IF K$="#" THEN T%=1:RETURN:REM KOZVETLEN CIMZES
2490 IF K$="(" THEN 2600:REM INDIREKT
2500 ZP=0:K$="*":REM NULL-AS LAP
2510 Z$=MID$(Y$,2+ZP)
2520 IF LEN(Z$)<2 THEN 2550
2530 K$=MID$(Z$,LEN(Z$)-1,1)
2540 IF K$="," THEN 2570:REM INDEXELT
2550 T%=5
2560 T%=T%+3*ZP:RETURN:REM ABSZOLUT BZW. NULL-AS LAP
2570 K$=RIGHT$(Z$,1):IF K$="X" THEN T%=6:GOTO 2560
2580 IF K$="Y" THEN T%=7:GOTO 2560
2590 T%=-1:RETURN:REM SYNTAX ERROR
2600 K$=RIGHT$(Z$,1):IF K$=")" THEN 2630
2610 IF RIGHT$(Z$,2)<>",Y" THEN 2590
2620 T%=11:RETURN
2630 IF MID$(Z$,LEN(Z$)-2,2)=",X" THEN T%=10:RETURN
2640 T%=12:RETURN
2700 IFX$="="THEN2730:REM PSEUDO-OPS PASS 1
2710 IF LEFT$(X$,2)="*=" THEN 2780
2715 IF LEFT$(X$,3)=".BY" THEN A%=1:RETURN
2720 A%=0:RETURN
2730 A%=0:IF Y$="*" THEN RETURN
2740 A%=ASC(LEFT$(LN$,1)):IF A%<65 OR A%>90 THEN RETURN
2750 A$=LEFT$(Y$,1):IF A$<>"$" AND (A$<"0" OR A$>"9") THEN RETURN
2760 A$=Y$:GOSUB 3100:IF F% THEN ML$(HC%)=FL$(2):RETURN
2770 GOSUB 3240:HE$(HC%)=RIGHT$("0000"+A$,4):RETURN
2780 A%=0:Y1$=LEFT$(Y$,1):IF Y1$="$" OR Y1$>"/" AND Y1$<":" THEN 2800
2790 RETURN
2800 A$=Y$:GOSUB 3100:IF F% THEN RETURN
2810 HA=A:GOSUB 3240:X%=ASC(LEFT$(LN$+CHRS(0),1)):IFX%>64ANDX%<91THENHE$(HC%)=A$
2820 RETURN
2900 IF XX$="=" THEN 2940:REM PSEUDO-OBS PASS 2
2910 IF LEFT$(X$,2)="*=" THEN 2990
2915 IF LEFT$(X$,3)=".BY" THEN 2991
2920 FL$(1)="S"
2930 A%=0:F%=1:PR$="                              ":RETURN
2940 A%=0
2950 A$=LEFT$(Y$,1)
2960 IF A$<>"*" AND A$<>"$" AND (A$<"0" OR A$>"9") THEN FL$(2)="S":GOTO 2930
2970 SL$=LN$:F%=0:GOSUB 4500:IF F% THEN FL$(0)=FL$(2):FL$(2)=" ":GOTO 2930
2980 PR$=HE$+"                ":RETURN
2990 A%=0:YZ$=LEFT$(Y$,1):IF YZ$="$" OR YZ$>"/" AND YZ$<":" THEN 3010
2991 YZ$=LEFT$(Y$,1):LH%=YZ$=">"ORYZ$="<":YA$=MID$(Y$,1-LH%)
```

```
2992 YZ$=LEFT$(YA$,1):IF YZ$="$" OR YZ$>"/" AND Z$<":" THEN HE$=YA$:GOTO 2994
2993 SL$=YA$:F%=0:GOSUB 4500:HE$="$"+HE$:IF F% THEN FL$(0)=FL$(2):FL$(2)=" "
2994 A$=HE$:GOSUB 3100:IF A>LO AND LH%=0 THEN A=0:FL$(1)="0"
2995 IF LEFT$(Y$,1)=">" THEN A=INT(A/HI)
2996 IF LEFT$(Y$,1)="<" THEN A=A-INT(A/HI)*HI
2998 POKE AD,A:A%=1:GOSUB 3240:PR$=PR$+RIGHT$("00"+A$,2)+"            ":RETURN
3000 FL$(2)="S":F%=1:GOTO 3030
3010 A$=Y$:GOSUB 3100:IF F% THEN 3030
3020 AD=A:GOSUB 3240:PR$=A$+"  "
3030 PR$=PR$+"            ":RETURN
3100 REM WANDLUNG HEX -> DEC  A$ -> A
3110 Z$=LEFT$(A$,1):IF Z$="$" THEN A$=RIGHT$(A$,LEN(A$)-1):GOTO 3150
3120 IF Z$<"0" OR Z$>"9" THEN FL$(2)="S":F%=1:RETURN
3130 A=VAL(A$):IF A>65535 OR A<0 THEN FL$(2)="0":F%=1
3140 RETURN
3150 A=0:L%=LEN(A$):IF L%>4 THEN F%=1:FL$(2)="L":RETURN
3200 FOR I=1 TO L%:AA%=ASC(MID$(A$,I))-48
3210 IF AA%<0 OR AA%>9 THEN IF AA%<17 OR AA%>22 THEN F%=1:FL$(2)="S":RETURN
3220 IF AA%>9 THEN AA%=AA%-7
3230 A=A+AA%*16^(L%-I):NEXT:RETURN
3240 AH%=A/HI:AL%=A-AH%*HI:A$=A$(AH%/16)+A$(AH%AND15)+A$(AL%/16)+A$(AL%AND15)
3250 RETURN
4000 DIM LB$(349),HE$(349),ML$(349):HA=AD:REM CIMTABLAZAT FELEPITESE
4010 FOR I=0 TO 349:LB$(I)="     ":HE$(I)="0000":ML$(I)=" ":NEXT
4020 OP$=DD$+":"+SN$+".SRC"
4030 OPEN 8,DG,0,OP$
4040 REM GET#8,A$,A$:LL%=349
4050 IF ST<>0 THEN CLOSE 8:END
4060 GOSUB 2000:PRINT CHR$(145),ZN$:IF LN$="" OR LEFT$(LN$,1)=" " THEN 4210
4070 X%=ASC(LEFT$(LN$,1)):IF X%<65 OR X%>90 THEN 4210
4080 GOSUB 4100:GOTO 4130
4090 LN$=LEFT$(LN$+"     ",5):REM HASH-CODE BILDEN
4100 HC=0:FOR I=1 TO 5
4110 HC%=ASC(MID$(LN$,I,1)):HC=HC+(HC%/10-INT(HC%/10))*10^(6-I):NEXT I
4120 HC%=(HC/307-INT(HC/307))*300:RETURN
4130 A=HA:GOSUB 3240
4140 IF LB$(HC%)<>"     " THEN 4180
4150 LB$(HC%)=LN$:HE$(HC%)=A$:IF HC%>UL% THEN UL%=HC%
4160 IF HC%<LL% THEN LL%=HC%
4170 GOTO 4210
4180 IF LB$(HC%)=LN$ THEN ML$(HC%)="M":GOTO 4210
4190 HC%=HC%+1:IF HC%<350 THEN 4140
4200 PRINT "SYMBOL TABELLE VOLL":CLOSE 8:END
4210 IFX$=".EN" THEN CLOSE 8:RETURN
4220 XX$=LEFT$(X$,1):IFXX$="."ORXX$="*"ORXX$="="THENGOSUB2700:HA=HA+A%:GOTO4060
4230 F%=0:XX$=LEFT$(X$,3):FOR J=0 TO NN%:IF XX$<>MN$(J) THEN NEXT:GOTO 4270
4240 GOSUB 2400
4250 IF T%(J,T%)>=0 THEN 4280
4260 IF T%=5 AND T%(J,9)>=0 THEN T%=9:GOTO 4280
4270 F%=1:HA=HA+1:GOTO 4060
4280 HA=HA+L%(T%):GOTO 4060
4500 REM     *********  CIM KERESES
4510 X%=ASC(LEFT$(SL$,1)):IF X%<65 OR X%>90THENFL$(2)="S":F%=1:HE$="0000":RETURN
4520 IF LEN(SL$)>5 THEN FL$(2)="L"
4530 SV$=LN$:LN$=SL$:GOSUB 4090:SL$=LN$:LN$=SV$
4540 IF LB$(HC%)="     " OR HC%>UL% THEN FL$(2)="U":F%=1:HE$="0000":RETURN
4550 IF LB$(HC%)<>SL$ THEN 4580
4560 HE$=HE$(HC%):IF ML$(HC%)<>" " THEN FL$(2)=ML$(HC%)
4570 RETURN
4580 HC%=HC%+1:GOTO 4540
4590 Y1$="":Y2$="":I=1:REM Y$ HELYETTESITESE Y1$ ES Y2$-BA
4600 IF MID$(Y$,I,1)<>"," THEN Y1$=Y1$+MID$(Y$,I,1)
4610 IF I>LEN(Y$) THEN F%=1:RETURN
4620 IF MID$(Y$,I,1)<>"," THEN I=I+1:GOTO 4600
4630 I=I+1:IF I>LEN(Y$) THEN F%=1:RETURN
4640 Y2$=Y2$+MID$(Y$,I,1):IF I=LEN(Y$) THEN F%=0:RETURN
4650 I=I+1:GOTO 4640
5000 READ NN%:HI=256:LO=255
```

```
5010 DIM A$(15),MN$(NN%),T%(NN%,12),L%(12),FL$(3),NP$(3)
5020 FOR I=0 TO 15:READ A$(I):NEXT
5030 NP$(1)="00            ":NP$(2)="00 00     ":NP$(3)="00 00 00  "
5040 FOR I=0 TO 12:READ L%(I):NEXT
5050 FOR J=0 TO NN%:READ MN$(J):FOR JJ=0 TO12:READ A$:IFA$="-1"THENA=-1:GOTO5070
5060 A=0:FOR I=1 TO 2:X=ASC(RIGHT$(A$,I))-48:X=X+(X>9)*7:A=A+X*16^(I-1):NEXT
5070 T%(J,JJ)=A:NEXT:NEXT:RETURN
6000 DATA 55:REM A MNEMONIKOK SZAMA
6010 DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
6020 DATA 1,2,2,2,2,3,3,3,1,2,2,2,3
7000 DATA ADC,-1,69,65,75,-1,6D,7D,79,-1,-1,61,71,-1
7010 DATA AND,-1,29,25,35,-1,2D,3D,39,-1,-1,21,31,-1
7020 DATA ASL,0A,-1,06,16,-1,0E,1E,-1,-1,-1,-1,-1,-1
7030 DATA BCC,-1,-1,-1,-1,-1,-1,-1,-1,-1,90,-1,-1,-1
7040 DATA BCS,-1,-1,-1,-1,-1,-1,-1,-1,-1,B0,-1,-1,-1
7050 DATA BEQ,-1,-1,-1,-1,-1,-1,-1,-1,-1,F0,-1,-1,-1
7060 DATA BMI,-1,-1,-1,-1,-1,-1,-1,-1,-1,30,-1,-1,-1
7070 DATA BIT,-1,-1,24,-1,-1,2C,-1,-1,-1,-1,-1,-1,-1
7080 DATA BNE,-1,-1,-1,-1,-1,-1,-1,-1,-1,D0,-1,-1,-1
7090 DATA BPL,-1,-1,-1,-1,-1,-1,-1,-1,-1,10,-1,-1,-1
7100 DATA BRK,-1,-1,-1,-1,-1,-1,-1,-1,00,-1,-1,-1,-1
7110 DATA BVC,-1,-1,-1,-1,-1,-1,-1,-1,-1,50,-1,-1,-1
7120 DATA BVS,-1,-1,-1,-1,-1,-1,-1,-1,-1,70,-1,-1,-1
7130 DATA CLC,-1,-1,-1,-1,-1,-1,-1,-1,18,-1,-1,-1,-1
7140 DATA CLD,-1,-1,-1,-1,-1,-1,-1,-1,D8,-1,-1,-1,-1
7150 DATA CLI,-1,-1,-1,-1,-1,-1,-1,-1,58,-1,-1,-1,-1
7160 DATA CLV,-1,-1,-1,-1,-1,-1,-1,-1,B8,-1,-1,-1,-1
7170 DATA CMP,-1,C9,C5,D5,-1,CD,DD,D9,-1,-1,C1,D1,-1
7180 DATA CPX,-1,E0,E4,-1,-1,EC,-1,-1,-1,-1,-1,-1,-1
7190 DATA CPY,-1,C0,C4,-1,-1,CC,-1,-1,-1,-1,-1,-1,-1
7200 DATA DEC,-1,-1,C6,D6,-1,CE,DE,-1,-1,-1,-1,-1,-1
7210 DATA DEX,-1,-1,-1,-1,-1,-1,-1,-1,CA,-1,-1,-1,-1
7220 DATA DEY,-1,-1,-1,-1,-1,-1,-1,-1,88,-1,-1,-1,-1
7230 DATA EOR,-1,49,45,55,-1,4D,5D,59,-1,-1,41,51,-1
7240 DATA INC,-1,-1,E6,F6,-1,EE,FE,-1,-1,-1,-1,-1,-1
7250 DATA INX,-1,-1,-1,-1,-1,-1,-1,-1,E8,-1,-1,-1,-1
7260 DATA INY,-1,-1,-1,-1,-1,-1,-1,-1,C8,-1,-1,-1,-1
7270 DATA JMP,-1,-1,-1,-1,-1,4C,-1,-1,-1,-1,-1,-1,6C
7280 DATA JSR,-1,-1,-1,-1,-1,20,-1,-1,-1,-1,-1,-1,-1
7290 DATA LDA,-1,A9,A5,B5,-1,AD,BD,B9,-1,-1,A1,B1,-1
7300 DATA LDX,-1,A2,A6,-1,B6,AE,-1,BE,-1,-1,-1,-1,-1
7310 DATA LDY,-1,A0,A4,B4,-1,AC,BC,-1,-1,-1,-1,-1,-1
7320 DATA LSR,4A,-1,46,56,-1,4E,5E,-1,-1,-1,-1,-1,-1
7330 DATA NOP,-1,-1,-1,-1,-1,-1,-1,-1,EA,-1,-1,-1,-1
7340 DATA ORA,-1,09,05,15,-1,0D,1D,19,-1,-1,01,11,-1
7350 DATA PHA,-1,-1,-1,-1,-1,-1,-1,-1,48,-1,-1,-1,-1
7360 DATA PHP,-1,-1,-1,-1,-1,-1,-1,-1,08,-1,-1,-1,-1
7370 DATA PLA,-1,-1,-1,-1,-1,-1,-1,-1,68,-1,-1,-1,-1
7380 DATA PLP,-1,-1,-1,-1,-1,-1,-1,-1,28,-1,-1,-1,-1
7390 DATA ROL,2A,-1,26,36,-1,2E,3E,-1,-1,-1,-1,-1,-1
7400 DATA ROR,6A,-1,66,76,-1,6E,7E,-1,-1,-1,-1,-1,-1
7410 DATA RTI,-1,-1,-1,-1,-1,-1,-1,-1,40,-1,-1,-1,-1
7420 DATA RTS,-1,-1,-1,-1,-1,-1,-1,-1,60,-1,-1,-1,-1
7430 DATA SBC,-1,E9,E5,F5,-1,ED,FD,F9,-1,-1,E1,F1,-1
7440 DATA SEC,-1,-1,-1,-1,-1,-1,-1,-1,38,-1,-1,-1,-1
7450 DATA SED,-1,-1,-1,-1,-1,-1,-1,-1,F8,-1,-1,-1,-1
7460 DATA SEI,-1,-1,-1,-1,-1,-1,-1,-1,78,-1,-1,-1,-1
7470 DATA STA,-1,-1,85,95,-1,8D,9D,99,-1,-1,81,91,-1
7480 DATA STX,-1,-1,86,-1,96,8E,-1,-1,-1,-1,-1,-1,-1
7490 DATA STY,-1,-1,84,94,-1,8C,-1,-1,-1,-1,-1,-1,-1
7500 DATA TAX,-1,-1,-1,-1,-1,-1,-1,-1,AA,-1,-1,-1,-1
7510 DATA TAY,-1,-1,-1,-1,-1,-1,-1,-1,A8,-1,-1,-1,-1
7520 DATA TSX,-1,-1,-1,-1,-1,-1,-1,-1,BA,-1,-1,-1,-1
7530 DATA TXA,-1,-1,-1,-1,-1,-1,-1,-1,8A,-1,-1,-1,-1
7540 DATA TXS,-1,-1,-1,-1,-1,-1,-1,-1,9A,-1,-1,-1,-1
7550 DATA TYA,-1,-1,-1,-1,-1,-1,-1,-1,98,-1,-1,-1,-1
```

A 6510-es Assembler leírása és fontosabb változói:

100–190 Az Assembler kiírja a fejlécet, bekéri a forrásprogram nevét, majd megkérdezi, hogy kérünk-e assembler listát, ha igen, beolvassa az egységszámot. Válaszunknak megfelelően beállítja a PM, ill. a DG (egységszám) változók értékét. A változók kezdőértékeinek beállítására meghívja az 5000-es sorbeli rutint.

200–460 A program fóciklusa. Az első menetet (PASS 1) a 400-as rutin hajtja végre. Az Assembler az első menet után megnyitja a forrásprogramot olvasásra (adatfileként). Ha nem kértünk programlistát, a 250-es sorban csak az utasítások sorszámait (ZN$) írja ki a képernyőre. Megvizsgálja, hogy a soronkövetkező utasítás pszeudoutasítás vagy .EN, ami a program végét jelzi.
A PR$ változó tartalmazza a kinyomtatandó szöveget.
A 320-as sorban az Assembler ellenőrzi, hogy érvényes-e az utasítás. Ha nem, a hibát jelző kapcsoló értékét beállítja, és a gépi kódú programba utasításkódként nullát (BRK) ír.
A 350-es sorban elugrik a címzésmódot megvizsgálni (GOSUB 2400).
A 360-as sorban ismét meghív egy rutint a címzésmódtól függően, és ebben a rutinban meghatározza az utasítás hosszát, az utasítás kódját, és építi tovább a kiírandó füzért.
A 370-es sorban az utasítás kódját beírja a tárba. A 380-as sorban lépteti a programszámlálót. A 400-as sortól a 460-as sorig előkészíti a nyomtatást.
Ha az utasítás hibás volt, megnöveli a hibák számát tartalmazó változó értékét.
A 450-es sorban kiírja az aktuális sort.

500–520 Az egy byte-os utasítások kezelése. Az A% változó a byte-ok számát, a T% változó pedig az utasításkódot tartalmazza. Ha T% értéke nulla, az adott utasítás ezzel a címzésmóddal nem létezik. Ebben az esetben az Assembler ugrik az 1510-es sorra. Egyébként az utasítás kódját átváltja hexadecimálissá, és tovább építi a nyomtatandó füzért.

600–740 A két byte-os utasítások kezelése. A 610-es sorban megvizsgálja a műveletkódot, a 620-as sorban építi a füzért. Keresi a >, a < jeleket, ill. a • jelet, ami nulláslap címzésre utal. A 660-as sorban megvizsgálja, hogy a változó tartalma numerikuse. Ha nem, a 670-es sorban (GOSUB 4500) beolvassa a címkét. Az operandus értékét hexadecimálisra váltja. A 700-, 710-es sorokban végrehajtja a >, ill. < műveleteket. Végül a 720-as sorban megvizsgálja, hogy a kapott érték nagyobb-e, mint 255. A kapott értéket elhelyezi a tárba, és a füzért tovább építi.

800–890 A három byte-os utasítások kezelése a tárban; először az operandus alsó, majd felső byte-ját helyezi el.

900–990 A relatív címzés kezelése. A 970-es sorban kiszámítja a relatív címet, és megvizsgálja, hogy az beleesik-e az érvényes tartományba. Ha nem, R betű kiírásával jelzi a hibát, és a valódi offset helyett nullát ír. A 980-as sorban meghatározza a negatív szám kettes komplemensét. A kapott értéket ismét elhelyezi a tárban, és hozzáfűzi a kiírandó szöveghez.

1000–1420 Az Assembler program futásának befejezése. A program megkérdezi, hogy a kapott gépi kódú programot tárolja-e lemezen. Ha igennel válaszolunk, a gépi

kódú program nevét, kezdő- és végcímét is elhelyezi a tárban, majd meghívja az operációs rendszer SAVE rutinját.
A tár tartalmát és a tárolt program hosszát kiírja képernyőre.
Megkérdezi, hogy kérünk-e szimbólumtáblázatot, ha igen, az 1200-tól 1400-ig terjedő sorokban kiírja.
Ezzel az Assembler program futása befejeződik.

A főprogram által meghívott rutinok:

1500–1520 Hiba esetén egy, két, ill. három nulla byte-ot ad vissza az utasítás kódja helyett.

1600–1640 A rutin megvizsgálja, hogy a programsorban szereplő első szó címke vagy utasítás. Ha ez utóbbi, akkor az I változó értéke 0 és NN% értéke közé kell, hogy essen, ahol NN% az utasításszavak száma.

2000–2300 Ez a rutin beolvas egy programsort a lemezről, és kijelöli a sorszámhoz, a címkéhez, az utasításszóhoz, ill. az operandushoz rendelt változót. A 2300-as sorban beolvas egy byte-ot a lemezről, és azt a Z$ változóban tárolja. Ha ez nullabyte, beállítja az FF% kapcsoló értékét, amely a sor végét jelzi. Az első két byte a lemez láncolási címe, ezt az olvasás során kihagyja.
Ha a Z$ tartalma két nulla, akkor elérte a forrásprogram végét (.EN), egyébként ez a két byte az utasítás sorszáma. A rutin az ezt követő karaktereket addig olvassa, amíg üres karaktert nem talál, vagy el nem érte a sor végét. Ha az olvasott karakter pontosvessző, a maradék szöveget megjegyzésnek tekinti. A rutin az utasítás sorszámát a ZN$, a címkét az LN$, az utasításszót az X$, az operandust az Y$, végül a megjegyzést az RM$ változóban adja vissza a főprogramnak.

2400–2640 Ez a rutin meghatározza az utasításban használt címzésmódot. Először megvizsgálja, hogy szerepel-e az utasításban zárójel, vessző, X, ill. Y, *, ill. a közvetlen címzés jele a #.
A rutin lefutása után a T% változó tartalma jelzi a címzésmódot (értéke 0 és 12 közé esik). Ha az eredmény negatív vagy nagyobb mint 12, a címzésmód érvénytelen.

2700–2820 Ez a rutin kezeli az =, a *=, ill. a .BY pszeudoutasításokat.
A főprogram az első menetben hívja.

2900–2998 Ugyanazt a feladatot látja el, mint az előző rutin, a második menetben. Meghatározza a .BY utasítások kódját és előkészíti a nyomtatást.

3000–3030 Meghívja a számok átváltását végző rutinokat, és a kiirandó szövegbe elhelyezi a programszámláló aktuális értékét.

3100–3230 Az A$-ban lévő hexadecimális számot átváltja decimálisra, és a kapott értéket visszaadja az A változóban.

3240–3250 Ua. mint fent, megfordítva.
Hívás után az AL% és AH% változókban visszaadja az alsó, ill. felső byte értékét.

4000–4280	A PASS 1 végrehajtása. A rutin megkeresi a forrásprogram összes címkéjét és megállapítja pontos értéküket. A címkéket elhelyezi (a későbbi gyors visszakeresés miatt) az LB$ ( ) tömbbe, a megfelelő értékeket pedig a HE$ ( ) tömbbe. A rutinnak szüksége van az éppen vizsgált utasítás hosszára és a címzésmódra. Az utasításszámláló értékét aszerint növeli, hogy milyen értéket talált a T% változóban (címzésmód), illetve az L% ( ) tömbben, amely a kívánt címzésmódhoz tartozó hosszt adja meg.
4500–4650	A főprogram ezt a rutint a második menetben hívja. Az SL$ változóban megadott címke értéke. Ha a címkét nem találja, hibajelzést, egyébként egy hexadecimális értéket ad vissza a HE$ változóban.
5000–5070	Ez a rutin beállítja a DATA utasításban felsorolt tömbök kezdőértékét.
6000–7550	Az alapadatokat tartalmazó DATA utasítások. Tartalmazzák a címzésmódokhoz tartozó utasításhosszakat, utasításszavakat, kódokat és a címzésmódot.

A legfontosabb változók jelentése

SN$	A forrásprogram neve (az .SRC végződés nélkül). A létrehozott gépi kódú modult hasonló néven ".OBJ"-vel ellátva tároljuk.
DD$	Lemezszám
PG	Az output egység száma: 3-képernyő, 4-nyomtató
PM	Kapcsoló a nyomtatáshoz
A	Aktuális címérték
AD	A fordítás során az utasításszámláló pillanatértéke
ZN$	Utasítássorszám
LN$	Címkenév
X$	Utasításszó
Y$	Operandus
RM$	Megjegyzés
T%	Címzésmód (0–12)
OF	A létrehozott kód relatív címe tároláskor (ha nem használjuk, értéke 0)
A%	Az aktuális cím A, hexadecimális értéke
SL$	Címkereső (4500-as sor). A keresett címkék nevét tartalmazó tömb
HE$	A címkék hexadecimális értéke
Lo	255 (konstans)
Hi	256 (konstans)
DF	A címkülönbség relatív címzésnél
BS%	Hibás utasítások számlálása
MX	A címkék száma (soronként) a szimbólumtáblázat elkészítéséhez
PR$	Egy nyomtató sort tartalmazó füzér listázáshoz
MN$	Utasításszó (mnemonik)
Z$	A lemezről beolvasott karakter
MN%	Az utasításszavak (mnemonikok) száma
X%	ASCII kódok
ZP	A nulláslap címzést jelölő kapcsoló
HA	Egy címke értéke (PASS 1)

F%,FF%	Hibakapcsolók
HC,HC%	Hashkódok
FL$(3)	Hibakódok
LB$(349)	A címkékhez tartozó hexadecimális értékek
T%(55,12)	A címzésmódokat és műveletkódokat tartalmazó tömb. Az első index jelöli az utasításszót, a második a címzésmódot.
MN$(55)	Az utasításszavak táblázata abc sorrendben

6. EGY EGYLÉPÉSES SZIMULÁTOR

A programok tesztelése és a hibakeresés az egyik legnehezebb programozói munka. Ebben a fejezetben bemutatunk egy olyan BASIC nyelvű programot, amely ezt a feladatot jelentős mértékben megkönnyíti.
A program szimulálja a 6510-es processzor működését. RUN paranccsal elindítva folyamatosan kiírja a képernyőre a processzor regisztereinek tartalmát:

```
  PC AC XR YR SR SP NV-BDIZC
0000 00 00 00 20 FF 00100000
```

A regisztereket már ismerjük:

PC	Programszámláló (program counter)
AC	Akkumulátor (accu)
XR	X regiszter
YR	Y regiszter
SR	Állapotregiszter (status)
SP	Veremmutató (stack pointer)
N	Negatív kapcsoló (negative flag)
Y	Break kapcsoló (break flag)
Z	Nulla kapcsoló (zero flag)
C	Átviteli kapcsoló (carry flag)

A program futás közben kiírja a képernyőre a regiszterek nevét és tartalmát. A fenti kiírás az induló állapotot mutatja.
A regisztereket egy billentyű megnyomásával módosíthatjuk. A régi tartalom megjelenik a képernyő alján, és a kurzor villog. Gépeljünk be érvényes hexadecimális számot, és nyomjuk meg a RETURN billentyűt. Az új sor a képernyő tetejére kerül, az adatbeviteli sor pedig törlődik. A kapcsolók értékeit a kezdőbetűik leütésével változtathatjuk meg:

P	Ha módosítjuk a programszámlálót, a regiszterek kijelzése alatt megjelenik az új tárcímen talált utasításkód assembler alakja.
A	A fentiekhez hasonlóan változtathatjuk meg az akkumulátor tartalmát. A RETURN leütése után a regiszterek neve alatt az új érték látható.
X	Az X regiszter tartalmának módosítása
Y	Az Y regiszter tartalmának módosítása
S	A veremmutató aktuális értéke megjelenik a képernyő alján az adatbeviteli sorban. Módosítása a fentiekhez hasonló
N	Leütésekor az N kapcsoló értéke ellenkezőjére vált. Ha 0 volt, 1 lesz, és megfordítva. Ezzel párhuzamosan az állapotregiszter tartalma is módosul.
V	Leütésekor a V kapcsoló értéke ellenkezőjére vált.
B	A break kapcsoló módosítása
D	A decimális kapcsoló értéke ellenkezőjére vált.
I	A megszakítási kapcsoló értéke ellenkezőjére vált.
Z	A zéró kapcsoló értéke ellenkezőjére vált.
C	Az átviteli kapcsoló értéke ellenkezőjére vált.

A program által használt legfontosabb billentyű a SPACE billentyű. Ha ezt a billentyűt leütjük, a gép végrehajtja és assembler alakban megjeleníti azt az utasítást, amelyre a programszámláló mutat.

Ahogyan a nevéből is kiderül, a program csak szimulálja az utasítások végrehajtását. A regiszterek és kapcsolók értéke pontosan úgy változik, mintha az utasítást a processzor hajtotta volna végre. A képernyőn látható az az utasítás is, amelyet a processzor következőként hajtana végre.

Az elmondottakra nézzünk egy konkrét példát:
Az egyszerűség kedvéért elemezzük az operációs rendszer egy programrészletét az egylépéses szimulátorral. Állítsuk az utasításszámláló értékét $A81D-re.
A képernyőn a következőket látjuk:

```
 PC     AC    XR    YR    SR    SP    NV–BDIZC
A81D    00    00    00    20    FF    00100000

A81D    38    SEC
```

Nyomjuk meg a SPACE billentyűt. A program szimulálja az utasítást, és a képernyő tartalma a következőképpen módosul:

```
 PC     AC    XR    YR    SR    SP    NV–BDIZC
A81E    00    00    00    21    FF    00100001

A81E    A5    2B    LDA $2B
```

A SEC utasítás 1-re állította a C kapcsolót. Az állapotregiszter értéke ennek megfelelően $21. A programszámláló értéke 1-gyel nőtt. A kapott címen egy betöltőutasítás áll. Nyomjuk meg ismét a SPACE billentyűt.

```
 PC     AC    XR    YR    SR    SP    NV–BDIZC
A820    01    00    00    21    FF    00100001

A820    E9    01    SBC #$01
```

Az akkumulátorba betöltődik a $2B tárcím tartalma, azaz 1. A Z és az N kapcsolók értéke változatlan marad. A programszámláló értéke 2-vel nő: $A820 lesz. Ezen a címen az SBC $01 utasítás áll, amely az akkumulátor tartalmát 1-gyel csökkenti.

```
 PC     AC    XR    YR    SR    SP    NV–BDIZC
A822    00    00    00    23    FF    00100011

A822    A4    2C    LDY $2C
```

Kivonás után az akkumulátorban 0 lesz. A Z kapcsoló értéke emiatt 1. A C kapcsoló értéke változatlanul 1, hiszen a kivonásnál nem volt alulcsordulás. A következő utasítás a $A822-es címen: LDY $2C. Végrehajtása után a kép:

PC	AC	XR	YR	SR	SP	NV–BDIZC
A824	00	00	08	21	FF	00100001

```
A824    B0    01    BCS   $A827
```

Az Y regiszter tartalma $08, és a Z kapcsoló törlődött. A következő utasítás a $A824 címen egy feltételes ugrás. Vajon végrehajtaná az ugrást a processzor? Az elágazás feltétele a C kapcsoló állapota. Mivel a C kapcsoló értéke 1, az elágazás feltétele teljesült. Nyomjuk meg ismét a SPACE billentyűt:

PC	AC	XR	YR	SR	SP	NV–BDIZC
A827	00	00	08	21	FF	00100001

```
A827    85    41    STA   $41
```

Az elágazás megtörtént, a programszámláló a $A287-es címre mutat. Az ugrás nem befolyásolta a kapcsolók értékét. A következő utasítás az akkumulátor tartalmát a $41-es címen tárolja.

PC	AC	XR	YR	SR	SP	NV–BDIZC
A829	00	00	08	21	FF	00100001

```
A829    84    42    STY   $42
```

Sem az STA, sem a soronkövetkező STY utasítás nem változtatja meg a kapcsolókat.

PC	AC	XR	YR	SR	SP	NV–BDIZC
A82B	00	00	08	21	FF	00100001

```
A82B    60          RTS
```

Ezen a ponton felfüggesztjük a szimulátorprogram futását. A program óriási előnye, hogy minden lépésben látjuk a képernyőn a processzor regisztereit. Megvizsgálhatjuk, hogy az egyes utasításokra a processzor hogyan reagál, hogyan változnak a regiszterek és a kapcsolók. A program használója a szimuláció bármely pontján visszaállíthatja a programszámláló értékét, ha az egyes részfolyamatokat többször szeretné átnézni.
Az eddig megismerteken kívül a szimulátorprogram további szolgáltatásokat is nyújt.
A „kurzor le" billentyűvel a programszámláló értékét növelhetjük, kihagyva utasításokat. Így addig lépegethetünk a gépi kódú programban, amíg az általunk érdekesebbnek ítélt részekhez nem értünk.
Bizonyos utasítások, pl. az STA vagy az INC tényleges végrehajtása elronthatná a rendszerváltozók értékét, és a gép esetleg működésképtelenné válna. Ezért a program a tár tartalmát csak látszólag módosítja.
Ha azonban a programozó biztos abban, hogy nem vét hibát, és fontosnak tartja, hogy az adott tárcím tartalma ténylegesen megváltozzék, nyomja meg az E billentyűt. Hatására a képernyőn megjelenik a VALÓDI VÉGREHAJTÁS? I üzenet. Most nyomjuk meg az I és a RETURN billentyűt, ha igennel, ill. az N és a RETURN billentyűt ha nemmel akarunk válaszolni.
Az M billentyű leütésével kiírathatjuk képernyőre egy tetszőleges tárcím tartalmát. Módosíthat-

juk is, de a módosítást a program csak akkor hajtja végre, ha előzetesen az E billentyűvel tényleges végrehajtást kértünk.
A következő példában a verem sajátos szerepére szeretnénk rávilágítani. Hajtsunk végre egy BRK utasítást: írjunk egy tárcímre 0-t, ami a BRK utasítás kódja. Indítsuk el RUN paranccsal a szimulátort, térjünk át E üzemmódra, majd a programszámláló értékét állítsuk $0002-re:

PC	AC	XR	YR	SR	SP	NV–BDIZC
0002	00	00	00	20	FF	00100000

```
0002      00      BRK
```

Az akkumulátorba, az X és az Y regiszterekbe írjunk különböző értékeket, hogy jobban tudjuk követni a változásokat.

PC	AC	XR	YR	SR	SP	NV–BDIZC
0002	22	44	88	20	FF	00100000

```
0002      00      BRK
```

Ha a $0002-es tárcímen nem BRK utasítás, azaz nem 0 kód van, üssük le az M betűt és írjuk be a 0002 címet. Az adatbeviteli sorban megjelenik a 2-es tárcím régi tartalma, ennek helyére írjunk $00-t. Nyomjuk meg a SPACE billentyűt, és figyeljük az eredményt:

PC	AC	XR	YR	SR	SP	NV–BDIZC
FF48	22	44	88	34	FC	00110100

```
FF48      48      PHA
```

A B és az I kapcsolók értéke 1 lett. A veremmutató 3-mal csökkent, $FF-ről $FC-re, mivel a programszámláló (két byte) és az állapotregiszter értéke a verembe került.
A programszámlálóba betöltődik a $FFFE és a $FFFF tárcímek tartalma, azaz $FF48. Ezen a címen egy PHA utasítás áll, amely az akkumulátor tartalmát a verembe helyezi.

PC	AC	XR	YR	SR	SP	NV–BDIZC
FF49	22	44	88	34	FB	00110100

```
FF49      8A      TXA
```

A veremmutató értéke 1-gyel csökkent. A következő utasítás az X regiszter tartalmát a verembe tölti:

PC	AC	XR	YR	SR	SP	NV–BDIZC
FF4A	44	44	88	34	FB	00110100

```
FF4A      48      PHA
```

A kapcsolók változatlanok, hiszen az akkumulátor tartalma nem nulla és nem negatív. Az akkumulátor tartalma ismét a verembe kerül.

PC	AC	XR	YR	SR	SP	NV–BDIZC
FF4B	44	44	88	34	FF	00110100

FF4B 98 TYA

A veremmutató ismét csökken. Az Y regiszter tartalma az akkumulátorba kerül. Az N kapcsoló értéke 1 lesz, mivel az Y regiszterben $7F-nél nagyobb számérték (negatív érték) áll.

PC	AC	XR	YR	SR	SP	NV–BDIZC
FF4C	88	44	88	B4	FA	10110100

FF4C 48 PHA

Az akkumulátor tartalma ismét a verembe került. A regiszterek tartalmának átmeneti megőrzésére azért van szükség, hogy visszatéréskor helyre lehessen állítani a processzor eredeti állapotát.
A szubrutinból való visszatérés, az RTI utasítás.

PC	AC	XR	YR	SR	SP	NV–BDIZC
FF4D	88	44	88	B4	F9	10110100

FF4D BA TSX

Most olyan programrészre ugrunk, amely a regiszterek eredeti tartalmát visszaállítja. Hogy a folyamatot világosan követni tudjuk, írjunk minden regiszterbe nullát. A programszámlálót állítsuk $EA81-re.

PC	AC	XR	YR	SR	SP	NV–BDIZC
EA81	00	00	00	B4	F9	10110100

EA81 68 PLA

Ez az utasítás visszatölt egy értéket a veremből az akkumulátorba.

PC	AC	XR	YR	SR	SP	NV–BDIZC
EA82	88	00	00	B4	FA	10110100

EA82 A8 TAY

Az akkumulátor tartalmát visszatöltjük az Y regiszterbe.

PC	AC	XR	YR	SR	SP	NV–BDIZC
EA83	88	00	88	B4	FA	10110100

EA83 68 PLA

Ismét betöltjük a verem legfelső byte-ját az akkumulátorba.

PC	AC	XR	YR	SR	SP	NV–BDIZC
EA84	44	00	88	34	FB	00110100

EA84 AA TAX

Valahányszor betöltünk egy értéket a veremből, a veremmutató 1-gyel nő. Most áttöltjük az akkumulátor tartalmát az X regiszterbe.

PC	AC	XR	YR	SR	SP	NV–BDIZC
EA85	44	44	88	34	FB	00110100

```
EA85    68      PLA
```

Végül az akkumulátor eredeti értéke is a helyére kerül.

PC	AC	XR	YR	SR	SP	NV–BDIZC
EA86	22	44	88	34	FC	00110100

```
EA86    40      RTI
```

A regiszterek és a veremmutató tartalma ugyanaz, ami eredetileg, a BRK utasítás végrehajtása után volt. A verem tartalmát mindig az írással ellentétes sorrendben kell visszaolvasni. Ez az ún. veremelv (last in-first out). Az RTI utasítás hatására visszatérünk az eredeti programrészre, és a végrehajtást ott folytathatjuk, ahol abbahagytuk.

PC	AC	XR	YR	SR	SP	NV–BDIZC
0005	22	44	88	20	FF	00100000

```
0005    91  B3      STA ($B3),Y
```

Most már az állapotregiszter tartalma is az eredeti, és a programszámláló a BRK utasítást követő utasításra mutat.
Az egylépéses szimulátor a programok tesztelésének ideális eszköze. Lépésről lépésre követhetjük, hogyan működik a processzor, hogy programunk valóban úgy dolgozik-e, ahogyan terveztük. A hibakeresés, ami egy gépi kódú programnál meglehetősen nehézkes, a szimulátorral gyerekjáték.
Különösen nagy segítség ez a kezdők számára, akik még nem ismerik ki magukat a különböző címzésmódokban.
A következő oldalakon közöljük a szimulátorprogram listáját, majd az egyes rutinok és a használt változók rövid leírását.
Útmutató a program begépeléséhez: ha a programsor nem férne el egy képernyősorban, használjuk az utasításszavak rövidített alakját, pl. 'goto' helyett írjunk 'gO'-t (g SHIFT-o).

```
0 REM **** P15. ****
1 :
2 :
100 PRINT"[illegible]        6510 EGYLEPESES SZIMULATOR
110 PRINT"        ---------------------------
120 PRINT"       ┌──────┬───────────────────┬──────────┐"
130 PRINT"       | PC   | AC XR YR SR SP    |NV-BDIZC |"
140 PRINT"       |      |                   |          |"
150 PRINT"       └──────┴───────────────────┴──────────┘"
160 FF=255:HI=256:UL=2↑16:SC=2↑15-1:SP=FF
170 DIM MN$(FF),AD(FF),OP(FF),SP(FF),H$(15)
180 FOR J=0 TO 15:READ H$(J):NEXT
190 FOR J=0 TO FF:READ MN$(J),OP(J),AD(J):NEXT
200 REM A REGISZTEREK TARTALMANAK KIIRASA
```

```
210 PRINT"█████████████";
215 IF PC>=UL THEN PC=PC-UL
220 A=PC:GOSUB 2290:PRINT"███";
230 A=AC:GOSUB 2320:PRINT"██";
240 A=XR:GOSUB 2320:PRINT"██";
250 A=YR:GOSUB 2320:PRINT"██";
255 GOSUB 900:REM SR
260 A=SR:GOSUB 2320:PRINT"██";
270 A=SP:GOSUB 2320:PRINT"███";
280 PRINTCHR$(48+N);
290 PRINTCHR$(48+V);
300 PRINT"1";
310 PRINTCHR$(48+B);
320 PRINTCHR$(48+D);
330 PRINTCHR$(48+I);
340 PRINTCHR$(48+Z);
350 PRINTCHR$(48+C)
360 PRINT"█████                    ██████████████████████"
400 GET T$:IF T$="" THEN 400
405 IF T$=" " THEN 1100 :REM SZIMULACIO
410 IFT$="P"THENPRINT"PC    ";:A=PC:GOSUB2290:INPUT"███████";A$:GOSUB2380:PC=A:GOTO
1000
420 IF T$="A" THEN T$="AC":A=AC:GOSUB 540:AC=A:GOTO 200
430 IF T$="X" THEN T$="XR":A=XR:GOSUB 540:XR=A:GOTO 200
440 IF T$="Y" THEN T$="YR":A=YR:GOSUB 540:YR=A:GOTO 200
450 IF T$="S" THEN T$="SP":A=SP:GOSUB 540:SP=A:GOTO 200
460 IF T$="N" THEN N=1-N:GOTO 200
470 IF T$="V" THEN V=1-V:GOTO 200
480 IF T$="B" THEN B=1-B:GOTO 200
490 IF T$="D" THEN D=1-D:GOTO 200
500 IF T$="I" THEN I=1-I:GOTO 200
510 IF T$="Z" THEN Z=1-Z:GOTO 200
520 IF T$="C" THEN C=1-C:GOTO 200
525 IF T$="█" THEN S=P:E=P:PC=P:GOTO 1010
527 IF T$="M" THEN 3000
528 IF T$="E" THEN 3100
530 GOTO 400
540 PRINTT$"   ";:GOSUB 2320:PRINT"   ";:INPUT"███████";A$:GOTO 2380
900 SR=N*128+V*64+32+B*16+D*8+I*4+Z*2+C:RETURN
910 N=SGN(SR AND 128):V=SGN(SR AND 64):B=SGN(SR AND 16):D=SGN(SR AND 8)
920 I=SGN(SR AND 4):Z=SGN(SR AND 2):C=SR AND 1:RETURN
980 N=SGN(AC AND 128):Z=1-SGN(AC):REM KAPCSOLOK
990 PC=PC+1+L
1000 S=PC:E=PC
1010 PRINT"██████████":GOSUB 2040:GOTO 200
1100 A=OP(PEEK(PC)):L=0:IF A=0 THEN 990
1110 ONAGOTO1200,1210,1220,1230,1240,1250,1260,1270,1280,1290,1300,1310,1320,133
0
1115 A=A-14
1120 ONAGOTO1340,1350,1360,1370,1380,1390,1400,1410,1420,1430,1440,1450,1460,147
0
1125 A=A-14
1130 ONAGOTO1480,1490,1500,1510,1520,1530,1540,1550,1560,1570,1580,1590,1600,161
0
1135 A=A-14
1140 ONAGOTO1620,1630,1640,1650,1660,1670,1680,1690,1700,1710,1720,1730,1740,175
0
1150 GOTO 200
1200 IF D THEN 1205:REM ADC
1201 GOSUB 1900:V=1-SGN(AC AND 128):AC=AC+OP+C:C=-(AC>FF)
1202 AC=AC AND FF:N=SGN(AC AND 128):V=VANDN:GOTO 980
1205 GOSUB 1900:AC=VAL(H$(AC/16)+H$(AC AND 15)):OP=VAL(H$(OP/16)+H$(OP AND 15))
1206 AC=AC+OP+C:C=-(AC>99):IF AC>99 THEN AC=AC-100
1207 A$=MID$(STR$(AC),2):GOSUB 2390:AC=A:GOTO 980
1210 REM AND
1211 GOSUB 1900:AC=AC AND OP:GOTO 980
1220 REM ASL
1221 IF AD(PEEK(PC))=4 THEN AC=AC*2:C=-(AC>FF):AC=AC AND FF:GOTO 980
1222 GOSUB 1900:A=OP*2:C=-(A>FF):A=A AND FF:GOSUB 1850
```

```
1223 N=SGN(OP AND FF):Z=1-SGN(OP):GOTO 990
1230 REM BCC
1231 FL=1-C:GOTO 1800
1240 REM BCS
1241 FL=C:GOTO 1800
1250 REM BEQ
1251 FL=Z:GOTO 1800
1260 REM BIT
1261 GOSUB 1900:N=SGN(OP AND 128):V=SGN(OP AND 64):Z=1-SGN(OP AND AC):GOTO 990
1270 REM BMI
1271 FL=N:GOTO 1800
1280 REM BNE
1281 FL=1-Z:GOTO 1800
1290 REM BPL
1291 FL=1-N:GOTO 1800
1300 REM BRK
1301 PC=PC+2:IF PC>=UL THEN PC=PC-UL
1302 PH=INT(PC/HI):PL=PC-PH*HI:SP(SP)=PH:SP=SP-1 AND FF:SP(SP)=PL:SP=SP-1 AND FF
1303 B=1:I=1:GOSUB 900:SP(SP)=SR:SP=SP-1 AND FF:PC=PEEK(UL-2)+HI*PEEK(UL-1):GOTO
 1000
1310 REM BVC
1311 FL=1-V:GOTO 1800
1320 REM BVS
1321 FL=V:GOTO 1800
1330 REM CLC
1331 C=0:GOTO 990
1340 REM CLD
1341 D=0:GOTO 990
1350 REM CLI
1351 I=0:GOTO 990
1360 REMCLV
1361 V=0:GOTO 990
1370 REM CMP
1371 GOSUB 1900:A=AC-OP
1372 N=SGN(A AND 128):Z=-(A=0):C=-(A>=0):GOTO 990
1380 REM CPX
1381 GOSUB 1900:A=XR-OP:GOTO 1372
1390 REM CPY
1391 GOSUB 1900:A=YR-OP:GOTO 1372
1400 REM DEC
1401 GOSUB 1900:A=OP-1 AND FF:GOSUB 1850
1402 GOTO 1442
1410 REM DEX
1411 XR=(XR-1) AND FF:GOTO 1452
1420 REM DEY
1421 YR=(YR-1) AND FF:GOTO 1452
1430 REM EOR
1431 GOSUB 1900:AC=(AC OR OP) AND NOT (AC AND OP)
1432 GOTO 980
1440 REM INC
1441 GOSUB 1900:A=OP+1 AND FF:GOSUB 1850
1442 N=SGN(A AND 128):Z=1-SGN(A):GOTO 990
1450 REM INX
1451 XR=(XR+1) AND FF
1452 Z=1-SGN(XR):N=SGN(XR AND 128):GOTO 990
1460 REM INY
1461 YR=(YR+1) AND FF
1462 Z=1-SGN(YR):N=SGN(YR AND 128):GOTO 990
1470 REM JMP
1471 GOSUB 1900:PC=AD:GOTO 1000
1480 REM JSR
1481 A=PC+2:PH=INT(A/HI):PL=A-PH*HI:SP(SP)=PH:SP=SP-1ANDFF:SP(SP)=PL:SP=SP-1ANDF
F
1482 PC=PEEK(PC+1)+PEEK(PC+2)*HI:GOTO 1000
1490 REM LDA
1491 GOSUB 1900:AC=OP:GOTO 980
1500 REM LDX
```

```
1501 GOSUB 1900:XR=OP:GOTO 1452
1510 REM LDY
1511 GOSUB 1900:YR=OP:GOTO 1462
1520 REM LSR
1521 IF AD(PEEK(PC))<>4 THEN 1524
1522 AC=AC/2
1523 C=-(AC<>INT(AC)):AC=AC AND FF:GOTO 960
1524 GOSUB 1900:A=OP/2:C=-(A<>INT(A)):A=A AND FF:GOSUB 1650
1525 GOTO 1442
1530 REM NOP
1531 GOTO 990
1540 REM ORA
1541 GOSUB 1900:AC=AC OR OP:GOTO 980
1550 REM PHA
1551 SP(SP)=AC:SP=SP-1 AND FF:GOTO 990
1560 REM PHP
1561 GOSUB 900:SP(SP)=SR:SP=SP-1 AND FF:GOTO 990
1570 REM PLA
1571 SP=(SP+1) AND FF:AC=SP(SP):GOTO 980:REM A KAPCSOLOK ALLITASA
1580 REM PLP
1581 SP=(SP+1) AND FF:SR=SP(SP):GOSUB 910:GOTO 990
1590 REM ROL
1591 IF AD(PEEK(PC))=4 THEN AC=AC*2+C:GOTO 1523
1592 GOSUB 1900:A=OP*2+C:C=-(A>FF)
1593 A=A AND FF:GOSUB 1850
1594 GOTO 1442
1600 REM ROR
1601 IF AD(PEEK(PC))=4 THENAC=AC/2+128*C:GOTO 1523
1602 GOSUB 1900:A=OP/2+128*C:C=-(A<>INT(A)):GOTO 1593
1610 REM RTI
1611 SP=SP+1 AND FF:SR=SP(SP):GOSUB 910:GOTO 1621
1620 REM RTS
1621 SP=SP+1 AND FF:A=SP(SP):SP=SP+1 AND FF:PC=A+SP(SP)*HI:GOTO 990
1630 IF D THEN 1635:REM SBC
1631 GOSUB 1900:V=SGN(AC AND 128):AC=AC-OP-1+C:C=-(AC>=0)
1632 AC=AC AND FF:N=SGN(AC AND 128):V=V AND 1-N:GOTO 980
1635 GOSUB 1900:AC=VAL(H$(AC/16)+H$(AC AND 15)):OP=VAL(H$(OP/16)+H$(OP AND 15))
1636 AC=AC-OP+C-1:C=-(AC>=0):IF AC<0 THEN AC=AC+100
1637 A$=MID$(STR$(AC),2):GOSUB 2390:AC=A:GOTO 980
1640 REM SEC
1641 C=1:GOTO 990
1650 REM SED
1651 D=1:GOTO 990
1660 REM SEI
1661 I=1:GOTO 990
1670 REM STA
1671 GOSUB 1900:A=AC:GOSUB 1850
1672 GOTO 990
1680 REM STX
1681 GOSUB 1900:A=XR:GOSUB 1850
1682 GOTO 990
1690 REM STY
1691 GOSUB 1900:A=YR:GOSUB 1850
1692 GOTO 990
1700 REM TAX
1701 XR=AC:GOTO 1452
1710 REM TAY
1711 YR=AC:GOTO 1462
1720 REM TSX
1721 XR=SP:GOTO 1452
1730 REM TXA
1731 AC=XR:GOTO 980
1740 REM TXS
1741 SP=XR:GOTO 990
1750 REM TYA
1751 AC=YR:GOTO 980
1800 REM ELAGAZASI PARANCSOK
```

```
1810 IF FL=0 THEN L=1:GOTO 990
1820 GOSUB 1985:GOTO 1000
1650 REM POKE
1670 IF AD<HI OR AD>HI+FF THEN 1880
1675 SP(AD-HI)=A:RETURN
1880 IF ES THEN POKE AD,A
1885 RETURN
1900 REM OPERANDUS BETOLTES
1910 A=AD(PEEK(PC))
1920 ON A GOSUB 1930,1935,1940,1945,1950,1955,1960,1965,1970,1975,1980,1985,1990
1925 IF AD<HI OR AD>HI+FF THEN RETURN
1927 OP=SP(AD-HI):RETURN
1930 AD=0:RETURN:REM MAGABAFOGLALT
1935 AD=PC+1:OP=PEEK(AD):L=1:RETURN:REM #
1940 AD=PEEK(PC+1):OP=PEEK(AD):L=1:RETURN:REM NULLASLAP
1945 AD=0:RETURN:REM A
1950 AD=PEEK(PC+1)+HI*PEEK(PC+2):OP=PEEK(AD):L=2:RETURN:REM ABSZOLUT
1955 AD=PEEK(PC+1)+XR AND FF:OP=PEEK(AD):L=1:RETURN:REM NULLASLAP,X
1960 AD=PEEK(PC+1)+YR AND FF:OP=PEEK(AD):L=1:RETURN:REM NULLASLAP,Y
1965 AD=PEEK(PC+1)+HI*PEEK(PC+2)+XR:OP=PEEK(AD):L=2:RETURN:REM ABSZOLUT,X
1970 AD=PEEK(PC+1)+HI*PEEK(PC+2)+YR:OP=PEEK(AD):L=2:RETURN:REM ABSZOLUT,Y
1975 AD=PEEK(PEEK(PC+1))+HI*PEEK(PEEK(PC+1)+1ANDFF)+YR:OP=PEEK(AD):L=1:RETURN:RE
M INDIREKT Y
1980 AD=PEEK(PC+1)+XR AND FF:AD=PEEK(AD)+HI*PEEK(AD+1):OP=PEEK(AD):L=1:RETURN:RE
M INDIREKT X
1985 A=PEEK(PC+1):A=A+HI*(A>127)+2+PC
1986 PC=INT(A/HI)*HI+((A+(A>SC)*UL) AND FF):RETURN:REM RELATIV
1990 AD=PEEK(PC+1)+HI*PEEK(PC+2):AD=PEEK(AD)+HI*PEEK(AD+1):OP=PEEK(AD):RETURN
2040 FOR P=S TO E:PRINT" ";
2050 A=P:GOSUB 2290:REM CIM
2060 PRINT" ";:A=PEEK(P):GOSUB 2320:PRINT" ";:J=PEEK(P):OP=AD(J)
2070 ON OP GOSUB 2350,2360,2360,2350,2370,2360,2360,2370,2370,2360,2360,2360,237
0
2080 PRINT" ";MN$(J)" ";
2090 ON OP GOSUB2110,2120,2130,2140,2150,2160,2170,2180,2190,2200,2210,2220,2240
2100 PRINT"   ":NEXTP
2105 IF P>=UL THEN P=P-UL
2110 RETURN
2120 PRINT"#";:GOSUB 2330:P=P+1:RETURN
2130 GOSUB 2330:P=P+1:RETURN
2140 PRINT" A";:RETURN
2150 GOSUB 2260:P=P+2:RETURN
2160 GOSUB 2330:P=P+1:PRINT",X";:RETURN
2170 GOSUB 2330:P=P+1:PRINT",Y";:RETURN
2180 GOSUB 2260:P=P+2:PRINT",X";:RETURN
2190 GOSUB 2260:P=P+2:PRINT",Y";:RETURN
2200 PRINT"(";:GOSUB 2330:P=P+1:PRINT"),Y";:RETURN
2210 PRINT"(";:GOSUB 2330:P=P+1:PRINT"),X";:RETURN
2220 A=PEEK(P+1):A=A+HI*(A>127)+2+P
2230 A=INT(A/HI)*HI+((A+(A>SC)*UL)ANDFF):PRINT"$";:GOSUB 2290:P=P+1:RETURN
2240 PRINT"(";:GOSUB 2260
2250 PRINT")";:P=P+2:RETURN
2260 PRINT"$";
2270 A=PEEK(P+1)+HI*PEEK(P+2)
2280 REM HEXACIM A
2290 HB=INT(A/HI):A=A-HI*HB
2300 PRINTH$(HB/16)H$(HBAND15);
2310 REM HEXABYTE A
2320 PRINTH$(A/16)H$(A AND 15);:RETURN
2330 PRINT"$";
2340 A=PEEK(P+1):GOTO 2320
2350 PRINT"      ";:RETURN
2360 GOSUB 2340:PRINT"   ";:RETURN
2370 GOSUB 2340:PRINT" ";:A=PEEK(P+2):GOTO 2320
2380 IF ASC(A$)=42 THEN END
2390 A=0:FOR J=1 TO LEN(A$):X=ASC(RIGHT$(A$,J))-48:X=X+(X>9)*7:A=A+X*(16↑(J-1))
2391 NEXT:RETURN
3000 PRINT:PRINT"**":PRINT"CIM:    $*****■■■■■■";:INPUT A$:GOSUB 2380
```

```
3010 PRINT"□",,:AD=A:OP=PEEK(A):GOSUB 1925:A=OP:GOSUB 2320:INPUT"████";A$:GOSUB
2380
3020 GOSUB 1850:PRINT"□                                         ":IF AD=PC THEN 1000
3030 GOTO 200
3100 INPUT"SZIMULACIO    I███";ES$:ES=ES$="I":GOTO 200
10000 DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E.F
10010 DATA "BRK",11,1,"ORA",35,11,"???",0,1
10020 DATA "???",0,1,"???",0,1,"ORA",35,3
10030 DATA "ASL",3,3,"???",0,1,"PHP",37,1
10040 DATA "ORA",35,2,"ASL",3,4,"???",0,1
10050 DATA "???",0,1,"ORA",35,5,"ASL",3,5
10060 DATA "???",0,1,"BPL",10,12,"ORA",35,10
10070 DATA "???",0,1,"???",0,1,"???",0,1
10080 DATA "ORA",35,6,"ASL",3.6,"???",0,1
10090 DATA "CLC",14,1,"ORA",35,9,"???",0,1
10100 DATA "???",0,1,"???",0,1,"ORA",35,8
10110 DATA "ASL",3,8,"???",0,1,"JSR",29,5
10120 DATA "AND",2,11,"???",0,1,"???",0,1
10130 DATA "BIT",7,3,"AND",2,3,"ROL",40,3
10140 DATA "???",0,1,"PLP",39,1,"AND",2,2
10150 DATA "ROL",40,4,"???",0,1,"BIT",7,5
10160 DATA "AND",2,5,"ROL",40,5,"???",0,1
10170 DATA "BMI",8,12,"AND",2,10,"???",0,1
10180 DATA "???",0,1,"???",0,1,"AND",2,6
10190 DATA "ROL",40,6,"???",0,1,"SEC",45,1
10200 DATA "AND",2,9,"???",0,1,"???",0,1
10210 DATA "???",0,1,"AND",2,8,"ROL",40,8
10220 DATA "???",0,1,"RTI",42,1,"EOR",24,11
10230 DATA "???",0,1,"???",0,1,"???",0,1
10240 DATA "EOR",24,3,"LSR",33,3,"???",0,1
10250 DATA "PHA",36,1,"EOR",24,2,"LSR",33,4
10260 DATA "???",0,1,"JMP",28,5,"EOR",24,5
10270 DATA "LSR",33,5,"???",0,1,"BVC",12,12
10280 DATA "EOR",24,10,"???",0,1,"???",0,1
10290 DATA "???",0,1,"EOR",24,6,"LSR",33,6
10300 DATA "???",0,1,"CLI",16,1,"EOR",24,9
10310 DATA "???",0,1,"???",0,1,"???",0,1
10320 DATA "EOR",24,8,"LSR",33,8,"???",0,1
10330 DATA "RTS",43,1,"ADC",1,11,"???",0,1
10340 DATA "???",0,1,"???",0,1,"ADC",1,3
10350 DATA "ROR",41,3,"???",0,1,"PLA",38,1
10360 DATA "ADC",1,2,"ROR",41,4,"???",0,1
10370 DATA "JMP",28,13,"ADC",1,5,"ROR",41,5
10380 DATA "???",0,1,"BVS",13,12,"ADC",1,10
10390 DATA "???",0,1,"???",0,1,"???",0,1
10400 DATA "ADC",1,6,"ROR",41,6,"???",0,1
10410 DATA "SEI",47,1,"ADC",1,9,"???",0,1
10420 DATA "???",0,1,"???",0,1,"ADC",1,8
10430 DATA "ROR",41,8,"???",0,1,"???",0,1
10440 DATA "STA",48,11,"???",0,1,"???",0,1
10450 DATA "STY",50,3,"STA",48,3,"STX",49,3
10460 DATA "???",0,1,"DEY",23,1,"???",0,1
10470 DATA "TXA",54,1,"???",0,1,"STY",50,5
10480 DATA "STA",48,5,"STX",49,5,"???",0,1
10490 DATA "BCC",4,12,"STA",48,10,"???",0,1
10500 DATA "???",0,1,"STY",50,6,"STA",48,6
10510 DATA "STX",49,7,"???",0,1,"TYA",56,1
10520 DATA "STA",48,9,"TXS",55,1,"???",0,1
10530 DATA "???",0,1,"STA",48,8,"???",0,1
10540 DATA "???",0,1,"LDY",32,2,"LDA",30,11
10550 DATA "LDX",31,2,"???",0,1,"LDY",32,3
10560 DATA "LDA",30,3,"LDX",31,3,"???",0,1
10570 DATA "TAY",52,1,"LDA",30,2,"TAX",51,1
10580 DATA "???",0,1,"LDY",32,5,"LDA",30,5
10590 DATA "LDX",31,5,"???",0,1,"BCS",5,12
10600 DATA "LDA",30,10,"???",0,1,"???",0,1
10610 DATA "LDY",32,6,"LDA",30,6,"LDX",31,7
```

```
10620 DATA "???",0,1,"CLV",17,1,"LDA",30,9
10630 DATA "TSX",53,1,"???",0,1,"LDY",32,8
10640 DATA "LDA",30,8,"LDX",31,9,"???",0,1
10650 DATA "CPY",20,2,"CMP",18,11,"???",0,1
10660 DATA "???",0,1,"CPY",20,3,"CMP",18,3
10670 DATA "DEC",21,3,"???",0,1,"INY",27,1
10680 DATA "CMP",18,2,"DEX",22,1,"???",0,1
10690 DATA "CPY",20,5,"CMP",18,5,"DEC",21,5
10700 DATA "???",0,1,"BNE",9,12,"CMP",18,10
10710 DATA "???",0,1,"???",0,1,"???",0,1
10720 DATA "CMP",18,6,"DEC",21,6,"???",0,1
10730 DATA "CLD",15,1,"CMP",18,9,"???",0,1
10740 DATA "???",0,1,"???",0,1,"CMP",18,8
10750 DATA "DEC",21,8,"???",0,1,"CPX",19,2
10760 DATA "SBC",44,11,"???",0,1,"???",0,1
10770 DATA "CPX",19,3,"SBC",44,3,"INC",25,3
10780 DATA "???",0,1,"INX",26,1,"SBC",44,2
10790 DATA "NOP",34,1,"???",0,1,"CPX",19,5
10800 DATA "SBC",44,5,"INC",25,5,"???",0,1
10810 DATA "BEQ",6,12,"SBC",44,10,"???",0,1
10820 DATA "???",0,1,"???",0,1,"SBC",44,6
10830 DATA "INC",25,6,"???",0,1,"SED",46,1
10840 DATA "SBC",44,9,"???",0,1,"???",0,1
10850 DATA "???",0,1,"SBC",44,8,"INC",25,8
10860 DATA "???",0,1
```

Az egylépéses szimulátorprogram leírása:

100–200	A regiszter-kijelző felépítése, a változók és a tömbök kezdeti értékének beállítása.
200–360	A regiszterek tartalmának kiírása. A regiszterek értékét kiírás előtt hexadecimális alakra konvertáljuk. A kapcsolók értékének megfelelően a képernyőre CHR$ függvénnyel 0-t vagy 1-et írunk.
400–530	A lenyomott billentyű vizsgálata. Ha a SPACE billentyű van lenyomva, az 1100-as sorra ugrunk, ahol az utasítás végrehajtását szimuláljuk. A regiszterek módosítását az adatbeviteli rutin végzi el. A régi értéket kiírja, és vár az új értékek begépelésére. Ha a „kurzor le" billentyű van lenyomva, a program a disassembláló rutinra ágazik el, és kiírja a következő utasítást.
900–920	Az SR státuszregiszter értékének kiszámítása a kapcsolók értékéből és megfordítva.
980	Az N és a Z kapcsolók értékének beállítása.
990	A programszámláló értékének növelése.
1000–1010	A következő utasítás disassemblálása.
1100–1150	Ugrás a szimulációra az utasítástól függően.
1200–1751	Az utasításokat szimuláló rutinok. A rutinok az utasítások abc sorrendjének megfelelően következnek egymás után. A szimuláció után a program a 990-es sorban az utasítás hosszától függően megnöveli a programszámláló értékét. A 980-as sorban módosítja az N és a Z kapcsolók tartalmát.

1800–1820	Az elágazó utasítások kezelése. A kapcsolók értékét az FL változóban tároljuk.
1850–1885	Ez a rutin beírja a tárba a kívánt értékeket. A veremtárat ($100-tól $1FF-ig) külön kezeli. A tényleges POKE utasítást csak akkor végzi el, ha E üzemmódban dolgozunk (az ES változó tartalma alapján).
1900–1990	A címzésmódnak megfelelő operandusok betöltése. Az operandus címe az AD, értéke az OP változóban van.
2040–2370	Ez a rutin tartalmaz egy disassembláló programrészt, amely a szimuláció után kiírja a soron következő utasítást. A 2070-es sorban a címzésmódtól függően kiírja az operandust. Ha a tárcím érvénytelen utasításkódot tartalmaz, az utasítás neve helyett egy kérdőjelet ír a képernyőre. A következő rutinok elvégzik a műveletet, és átváltják a számokat decimális alakról hexadecimális alakra.
2380–2390	A számok átváltása hexadecimális alakról decimális alakra. Ha szám helyett csillagot írunk be, a program futása befejeződik.
3000–3030	A tárcímek tartalmának kijelzése és megváltoztatása (az utóbbi csak E üzemmódban).
3100	Döntés az utasítások tényleges végrehajtásáról.
10000–10860	Az utasításszavakat, a kódokat, ill. a címzésmódokat tartalmazó DATA sorok. A program indításkor beolvassa ezeket a konstansokat a megfelelő tömbváltozókba.

A program változói:

FF	Konstans 255
HI	Konstans 256
UL	Konstans 65536
SC	Konstans 32767
MN$(255)	A 6510-es mnemonik tömbje
OP(255)	Az utasításkódok tömbje
D(255)	A címzésmódok tömbje
SP(255)	A veremtömb
H$(15)	A hexadecimális számjegyek tömbje
PC	A programszámláló
AC	Az akkumulátor
XR	Az X regiszter
YR	Az Y regiszter
SR	Az állapotregiszter
SP	A veremmutató
N	A negatív kapcsoló
V	A túlcsordulási kapcsoló
B	A break kapcsoló

D	A decimális kapcsoló
I	A megszakítási kapcsoló
Z	A zéró kapcsoló
C	Az átviteli kapcsoló
T$	A lenyomott billentyű
L	Az operandus hossza
ES	Az E üzemmód kapcsolója
OP	Az operandus

7. GÉPI KÓDÚ PROGRAMOZÁS A COMMODORE 64-ESEN

A gépi kódú programozás bemutatására különösen alkalmas terület ezen a számítógépen a nagyfelbontású grafika programozása. Ebben a fejezetben minden olyan rutint megírunk gépi kódban, amely a nagyfelbontású grafika programozása során szükséges.

Az Olvasó közben elsajátíthatja a processzor működésére vonatkozó alapismereteket, és a gépi kódú programozás technikáját.

A grafika programozása BASIC nyelven a POKE és a PEEK utasítások átláthatatlan szövevénye; ezen a területen különösen szembetűnő a gépi kódú programozás hatékonysága. Közben azt is bemutatjuk, hogy miként lehet a kétféle programnyelvet együtt használni úgy, hogy mindkét nyelv előnyeit kiaknázzuk. Amennyire szükséges, megismerkedünk a videovezérlő működésével. Ha az Olvasó mélyebb betekintést szeretne nyerni a Commodore 64-es hardverfelépítésébe, és további ismereteket kíván szerezni az operációs rendszerről, olvassa el „A Commodore 64-es belső felépítése"* c. könyvet.

A két programnyelv együttes használatának legfontosabb kérdése a paramétercsere. A gépi kódú programokat BASIC nyelvből SYS utasítással hívhatjuk meg, amelyben meg kell adni a hívott program tárbeli kezdőcímét. A gépi kódú program RTS utasítása visszaadja a vezérlést a BASIC programnak. Ha olyan gépi kódú programot hívtunk meg, amely egy feladatot hajt végre, pl. törli a képernyőt, nincs szükség paramétercserére. Ha azonban a gépi kódú rutin pl. egy pontot rajzol a képernyőre, paraméterként meg kell adnunk a pont helyzetét. A paraméterek átadásának többféle módja van:

Az egyik módszer az ún. levélszekrény módszer. A paramétereket, példánkban a vízszintes és függőleges koordinátát, elhelyezzük egy-egy tárcímre a BASIC POKE utasítással. A gépi kódú rutin ezeket be tudja olvasni, és fel tudja dolgozni. A BASIC interpreter más lehetőséget is kínál a paramétercserére.

A SYS utasítás végrehajtásakor átadhatjuk a processzornak bizonyos regiszterek tartalmát. Mivel azonban a regiszterek tartalmát BASIC utasítással nem tudjuk elérni (írni vagy olvasni), a rendszer szolgáltat erre a célra négy tárcímet. A következő tárcímek tartalma a SYS utasítás végrehajtásakor automatikusan áttöltődik a megfelelő regiszterekbe:

780 ⇒ akkumulátor
781 ⇒ X regiszter
782 ⇒ Y regiszter
783 ⇒ állapotregiszter

A gépi kódú program paramétereit POKE utasítással beírhatjuk a fenti tárcímekre. Az állapotregiszterrel óvatosan kell bánni, váratlan következményekkel járhat ugyanis, ha pl. megváltoztatjuk a decimális vagy a megszakítási kapcsoló értékét.

Mivel a tárcímek tartalma bekapcsoláskor 0, a rutin meghívásakor minden kapcsoló értéke törlődik, ha a BASIC utasítással a 783-as tárcím tartalmát nem változtattuk meg. Amikor a gépi kódú programból visszatérünk a BASIC programba, a regiszterek tartalma áttöltődik a fenti tárcímekre, ha szükséges, a továbbiakban ezeket ismét felhasználhatjuk. A paramétercserét más tárcímeken keresztül is megoldhatjuk, csak arra kell ügyelnünk, hogy a két program egymással összhangban hivatkozzon közös tárcímekre. A paramétercseréhez igénybe vehetjük a BASIC interpreter rutinjait is. Amikor az interpreter felismeri egy BASIC

* 64 Intern. DATA BECKER, 1983. Magyarul előkészületben (a szerk. megj.).

sorban a POKE utasítást, meghív egy olyan rutint, amely betölt egy paramétert a BASIC területről. A rutin ki tudja értékelni a POKE utasításban szereplő olyan bonyolult kifejezéseket is, mint pl. az alábbi:

POKE A+7,5*Z%(INT(SIN(X)*1000)),EXP(X)

A BASIC interpreternek ezt a rutinját hasznosíthatjuk a gépi kódú programokban is, de ezzel a módszerrel most nem foglalkozunk részletesen, egyenlőre alkalmazzuk a paramétercsere egyszerűbb, fent ismertetett módját. Mielőtt rátérnénk a gépi kódú programokra, tisztázzuk a nagyfelbontású grafika programozásának néhány alapelvét. Míg a karakteres grafikában a legkisebb programozható grafikus egység egy 8*8=64 képpontot tartalmazó négyzet, addig a nagyfelbontású képernyőn minden egyes pontra külön hivatkozhatunk.
A karakteres képernyő felbontása 25*40-es, a nagyfelbontású képernyő felbontása 200*
*320-as.
Mindkét ábrázolási módban a képernyőhöz hozzárendeljük az ún. *videoramot*. Ezen a tárterületen minden karakternek megfelel egy byte, ill. minden pontnak megfelel egy bit. A karakteres grafikához 25*40*1=1000 byte-ra, a nagyfelbontású grafikához pedig 200*320*1=64 000 bitre, azaz 8000 byte-ra van szükség. A karakteres ábrázolásnak megfelelő videoram az 1024-től 2023-ig ($400-tól, $7E8-ig) terjedő tárterület. A videoram kezdőcímét a video-vezérlő programozásával megváltoztathatjuk ($800-ra, $C00-ra, $1000-re stb.).
A nagyfelbontású grafika képernyőtára 8 kbyte-ot foglal el, amelyet szintén a video-vezérlő programozásával választhatunk ki. A tárterület kiválasztásánál ügyelni kell arra, hogy a video-vezérlő csak 16 kbyte-ot tud megcímezni. Hogy éppen melyik 16 kbyte-os egységre vonatkoznak a video-vezérlő címei, azt egy I/O kapcsoló dönti el.
Első pillantásra az tűnik legcélravezetőbbnek, hogy a 8 kbyte-ot a BASIC tárból vegyük el. Ennél azonban sokkal igényesebb megoldást is választhatunk, élve a gépi kódú programozás előnyeivel. A Commodore 64-es teljes tárterületét használhatjuk RAM-ként. Jelöljük ki a grafikus RAM-ot az operációs rendszer „alatt" a $E000-tól a $FFFF-ig terjedő tárterületen. Mivel ezt a területet közvetlenül BASIC-ből nem tudjuk olvasni, olvasáskor „ki kell kapcsolnunk" az interpretert és az operációs rendszert.
A video-vezérlő által címezhető 16 kbyte-os terület másik felét helyezzük a $C000-tól $C3FF-ig terjedő tárterületre, ezt a területet ugyanis a BASIC szintén nem használja.
A nagyfelbontású grafika szín-RAM-ként csak 1 kbyte-ot használhat, így nem lehet minden képernyőpont más színű. A karaktereknek megfelelő 8*8-as egységek azonos színűek lesznek.
Kezdjünk hozzá a gépi kódú rutinok megírásához. Elsőként készítsük el azt a rutint, amely a képernyőt átkapcsolja karakteres kijelzésről nagyfelbontású kijelzésre.
Fogalmazzuk meg először a feladatot BASIC nyelven:

```
0 REM **** P16. ****
1 :
2 :
100 V=53248 :REM VIDEOKONTROLL KEZDOCIME
110 V1=V+17 :REM ATKAPCSOLASI CIM GRAFIKUS MODRA
120 V2=V+24 :REM A VIDEORAM STARTCIMENEK ROGZITESE
130 CIA=$DD00 :REM 16 K LEFOGLALASA
140 POKE V1,59
150 POKE V2,8
160 POKE CIA,0
170 END
```

Ha át akarunk térni gépi kódra, el kell döntenünk, hogy a gépi kódú programot melyik tárterületre helyezzük. Mivel a $C000-tól $C400-ig terjedő területet elfoglalja a szín-RAM, legyen a gépi kódú program kezdőcíme: $C400.

Az utasítások átírása nem okoz gondot:

```
0 REM **** P17. ****
1 :
2 :
100 VIDEO=53248          ; VIDEOKONTROLL
110 V1=53625             ; GRAFIKUS MOD CIME
120 V2=53272             ; VIDEORAMCIM CIME
130 CIA=$DD00            ; 16 K KIVALASZTASA
140 *=$C400              ; A RUTIN KEZDOCIME
150 LDA #59
160 STA V1
170 LDA #8
180 STA V2
190 LDA #0
200 STA CIA
210 RTS
220 .EN
```

Fordítsuk le a programot az Assemblerrel:

```
0 REM **** P18. ****
1 :
2 :
D000                 100 VIDEO  =     53248
D011                 110 V1     =     53265
D018                 120 V2     =     53272
DD00                 130 CIA    =     $DD00
                     140        *=    $C400
C400  A9 3B          150 EIN    LDA   #59
C402  8D 11 D0       160        STA   V1
C405  A9 08          170        LDA   #8
C407  8D 18 D0       180        STA   V2
C40A  A9 00          190        LDA   #0
C40C  8D 00 DD       200        STA   CIA
C40F  60             210        RTS
                     220        .EN
```

Ahhoz, hogy a programot kipróbálhassuk, meg kell írnunk azt a programrészt is, amely a nagyfelbontású képernyőt visszakapcsolja normál üzemmódra, azaz a regiszterek eredeti értékét visszaállítja. A visszakapcsoló rutint helyezzük a program elejére:

```
0 REM **** P19. ****
1 :
2 :
100 VIDEO=53248        ;VIDEOKONTROLL
110 V1=53625           ;GRAFIKUS MOD CIME
120 V2=53272           ;VIDEORAMCIM CIME
130 CIA=$DD00          ;16 K KIVALASZTASA
140 *=$C400            ;A RUTIN KEZDOCIME
150 EIN LDA #59
160 STA V1
170 LDA #8
180 STA V2
190 LDA #0
200 STA CIA
```

```
210 RTS
220 ; KIKAPCSOLAS
230 AUS LDA #27
240 STA V1
250 LDA #21
260 STA V2
270 LDA #3
280 STA CIA
290 RTS
300 .EN
```

Assembláljuk ismét a programot, és kérjünk szimbólumtáblázatot. Annak ellenére, hogy a programban eddig még nem hivatkoztunk a KI és a BE címkékre, biztosan minden Olvasó kitalálja, hogy mi a célja ezeknek a címkéknek. A későbbiekben szükség lesz rájuk a SYS utasításokban.

```
0 REM **** P20 ****
1 :
2 :
  D000              100 VIDEO   =      53248    ; VIDEOKONTROLL
  D179              110 V1      =      53625    ; GRAFIKUS MOD CIME
  D018              120 V2      =      53272    ; VIDEORAMCIM CIME
  DD00              130 CIA     =      $DD00    ; 16 K KIVALASZTASA
  C400              140         *=     $C400    ; A RUTIN KEZDOCIME
  C400  A9 3B       150 EIN     LDA    #59
  C402  8D 79 D1    160         STA    V1
  C405  A9 08       170         LDA    #8
  C407  8D 18 D0    180         STA    V2
  C40A  A9 00       190         LDA    #0
  C40C  8D 00 DD    200         STA    CIA
  C40F  60          210         RTS
  C410              220                      ; KIKAPCSOLAS
  C410  A9 1B       230 AUS     LDA    #27
  C412  8D 79 D1    240         STA    V1
  C415  A9 15       250         LDA    #21
  C417  8D 18 D0    260         STA    V2
  C41A  A9 03       270         LDA    #3
  C41C  8D 00 DD    280         STA    CIA
  C41F  60          290         RTS
                    300         .EN

C400 / C420 / 0020
A FORRAS FILE: P20.SRC
 0  HIBA

AUS     C411    CIA     DD00    EIN     C400    V1      D179    V2      D018
VIDEO   D000
```

Számítsuk ki a címkék decimális értékét:
A BE és a KI címkék értéke $C400, azaz 50176, ill. $C410, azaz 50142.
Próbáljuk ki a gépi kódú rutint a következő BASIC programmal:

```
0 REM **** P21. ****
1 :
2 :
100 SYS 50176:REM A GRAFIKUS MOD BEKAPCSOLASA
110 GET A$:IF A$=""THEN 110
120 SYS 50192:REM A GRAFIKUS MOD KIKAPCSOLASA
```

A program átkapcsol grafikus üzemmódra, vár, amíg le nem ütünk egy billentyűt, majd visszakapcsol normál üzemmódra. Grafikus üzemmódban a képernyőn színes négyzetek kavalkádja látható. Ez a kép arra utal, hogy a gép bekapcsolásakor a használaton kívüli RAM területen véletlenszerű értékek találhatók.
A következő feladat a grafikus képernyő és a színtár törlése. BASIC nyelven a képernyőt egy POKE ciklussal törölhetnénk.
Ha a képernyő minden pontját törölni akarjuk, a képernyőtár minden bitjét 0-ra kell állítanunk. A ciklusnak a $E000-tól a $FFF-ig kell jutnia. (Egészen pontosan csak a $FF37-ig, hiszen nem 8192, hanem csak 8000 byte-ra van szükség).

```
FOR I=57344 TO 65535:POKE I,0:NEXT
```

A programsort BASIC nyelven rendkívül gyorsan meg tudtuk írni, a végrehajtása azonban annál tovább tart, kb. 30 másodpercig.
Az előző fejezetben már írtunk egy olyan gépi kódú ciklust, amely megjelenítette a képernyőn a C 64-es karakterkészletét.
Az a ciklus 256 lépést tartalmazott, erre elég volt az X és az Y regiszter. Most egy 8000 lépéses ciklust kell megírnunk. BASIC nyelven egymásba ágyazott ciklusokkal dolgoznánk:

```
0 REM **** P22. ****
1 :
2 :
AD = 57344
FOR X = 0 TO 31
FOR Y = 0 TO 255
POKE AD+Y, 0
NEXT Y
AD = AD+256
NEXT X
```

A 8192 byte-ot 32 lapra bonthatjuk fel, melyek egyenként 256 byte-ot tartalmaznak. Az Y regiszterrel vezérelt ciklussal 256 byte-ot tudunk törölni. Ha az AD báziscímet 256-tal megnöveljük, a következő 256 byte-ot töröljük. Mivel 32 lapot kell törölnünk, ezt a lépést 32-szer kell megismételni. A külső ciklust vezérelheti az X regiszter.
Az, hogy 192 byte-ot feleslegesen törlünk, nem okoz semmilyen problémát.
Az előtanulmányt kövesse a gépi kódú program:

```
0 REM **** P23. ****
1 :
2 :
100 AD = $E000
110 LDA #0 ; AKKU TOLTESE
120 LDX #0
130 LDY #0
140 STA AD,Y
150 INY
160 BNE SYMB1
170 ; AD = AD + $100
180 INX
190 ; AZ X VAN 31
200 ; NEM, AKKOR VISSZA 130-RA
210 .EN
```

Ez a program még nem egészen tökéletes. Valószínűleg kitalálta az Olvasó, hogy a SYMB1 címkét a 140-es sor elé kell helyezni.
A következő kérdés az lehet, hogy hogyan növeljük meg az AD változó tartalmát, hiszen amit a programba írtunk, az egyenlőre csak egy megjegyzés. Emlékezzünk vissza az indirekt indexelt címzésre! Az indirekt indexelt címzésben az aktuális címet a nulláslap 2 byte-os mutatójából és az Y regiszter tartalmából számítottuk ki. A mutató értékét minden lépésben növelhetjük $100-zal. Az egyszerű indexelt címzéssel csak egy 256 byte-os területet tudunk átfogni.
Nem okozhat gondot az X regiszter tartalmának ellenőrzése sem, amellyel az elágazást vezérelhetjük.

```
0 REM **** P24. ****
1 :
2 :
100 AD = $E000
110 LDA #0 ; AKKU TOLTESE
120 LDX #0
130 SYMB2 LDY #0
140 SYMB1 STA (AD),Y
150 INY
160 BNE SYMB1
170 ; AD = AD + $100
180 INX
190 CPX #32
200 BNE SYMB2
210 .EN
```

Az indirekt indexelt címzésben használt mutató két byte a nulláslapon. Használjuk erre a célra a $FA és $FB címeket. A mutató kezdőértéke legyen $E000, az alsó byte-ot ($00-t) töltsük a $FA, a felső byte-ot ($E0-t) töltsük a $FB címekre. A mutatót úgy tudjuk $100-zal megnövelni, hogy a felső byte értékét 1-gyel növeljük.
Fejezzük be a gépi kódú rutint az RTI utasítással:

```
0 REM **** P25. ****
1 :
2 :
90 * = $C420
100 LDA #<$E000
102 STA $FA
104 LDA #>$E000
106 STA $FB
110 LDA #0 ; AKKU TOLTESE
120 LDX #31
130 SYMB2 LDY #0
140 SYMB1 STA (AD),Y
150 INY
160 BNE SYMB1
170 INC $FB
180 INX
190 CPX #32
200 BNE SYMB2
205 RTS
210 .EN
```

Legyen a program kezdőcíme az előző két program utáni első szabad tárcím: $C420.
Tároljuk, majd assembláljuk a programot:

```
0 REM **** P26. ****
1 :
2 :
00FA                90 AD     =     $FA
00FB                95 AD1    =     $FB
C420                97        *=    $C420
C420  A9 00        100        LDA   #<$E000
C422  8D FA 00     102        STA   AD
C425  A9 E0        104        LDA   #>$E000
C427  8A FB 00     106        STA   AD1
C42A  A9 00        110        LDA   #0
C42C  A2 00        120        LDX   #0
C42E  A0 00        130 SYMB2  LDY   #0
C430  91 FA        140 SYMB1  STA   (AD),Y
C432  C8           150        INY
C433  D0 FB        160        BNE   SYMB1
C435  EE FB 00     170        INC   AD1
C438  E8           180        INX
C439  E0 20        190        CPX   #32
C43B  D0 F1        200        BNE   SYMB2
C43D  60           205        RTS
                   210        .EN
```

A FORRASPROGRAM: 2.PELDA.SRC
0 HIBA

Annak ellenére, hogy programozástechnikailag nem a legigényesebb módszereket választottuk, a fenti program mindenesetre futóképes.
Hogyan javíthatunk a szépséghibákon? Először is az AD és az AD1 címek helyett használhattunk volna nulláslap címzést. Ezt megtehetjük úgy, hogy eléjük írunk egy csillagot. Azután: fölöslegesen töltöttünk az akkumulátorba 0-t a 110-es sorban, hiszen ezt már megtettük a 100-as sorban is. Fordítsuk meg a 100-102. és a 104-106. utasítások sorrendjét!
Futtassuk az X regisztert 0-tól 32-ig! Így feleslegessé válik a 190-es sor. Végül a 130-as sorban az Y regiszterbe úgy töltünk 0-t, hogy egyszerűen beleírjuk az akkumulátor tartalmát.
A program ezekkel a javításokkal egy kicsit rövidebb lesz:

```
0 REM **** P27. ****
1 :
2 :
00FA                90 AD     =     $FA
00FB                95 AD1    =     $FB
C420                97        *=    $C420
C420  A9 E0        100        LDA   #>$E000
C422  85 FB        102        STA   *AD1
C424  A9 00        104        LDA   #<$E000
C426  85 FA        106        STA   *AD
C428  A2 20        110        LDX   #32
C42A  A8           120 SYMB2  TAY
C42B  91 FA        130 SYMB1  STA   (AD),Y
C42D  C8           140        INY
C42E  D0 FB        150        BNE   SYMB1
C430  E6 FB        160        INC   *AD1
C432  CA           170        DEX
C433  D0 F5        180        BNE   SYMB2
C435  60           190        RTS
                   200        .EN
```

Az alábbi BASIC programmal kipróbálhatjuk, hogyan működik a gépi kódú rutin módosított változata:

```
0 REM **** P28. ****
1 :
2 :
100 SYS 50176 :REM GRAFIKA BE
110 GET A$: IF A$="" THEN 110
120 SYS 50208 :REM GRAFIKA TORLESE
130 GET A$ :IF A$="" THEN 130
140 SYS 50192 :REM GRAFIKA KI
```

Meghívjuk az 50176-os címen kezdődő rutint, amely átkapcsolja a képernyőt grafikus üzemmódra. A következő sorban addig várakozunk, amíg a gépkezelő le nem üt egy billentyűt. Ekkor meghívjuk az 50208-as címen elhelyezett rutint, amely a másodperc törtrésze alatt törli a képernyőtárat. A képernyőtörlés kb. 30 másodpercig tartana BASIC nyelvű programmal! A következő billentyű leütése után ismét visszakapcsolunk normál üzemmódra.
Minden grafikát kezelő program indításkor inicializálja a képernyőt. Az inicializáláshoz a képernyőtár törlésén kívül a színeket is be kell állítanunk. Az eddigi tapasztalatok alapján ez sem nehezebb feladat mint az előző. Programozástechnikailag az egyetlen újdonságot az jelenti, hogy a háttér- és a megjelenő pontok színkódját paraméterként át kell adnunk a gépi kódú programnak. A két paramétert egy byte-on elhelyezhetjük: a felső félbyte-on a háttér színének, az alsó félbyte-on pedig a látható pontok színének kódját.
Amint azt már említettük, a képernyő minden 8 × 8 pontból álló egységéhez a színtárban 1 byte tartozik. Ha pl. a színtár valamely byte-ja $10-et tartalmaz, a felső félbyte értéke 1, az alsó félbyte értéke 0, tehát a byte-nak megfelelő négyzet háttérszíne fekete, a pontok színe pedig fehér.
Adjuk át a paramétereket az akkumulátoron keresztül! Javasoljuk, hogy az Olvasó először próbálja önállóan megoldani a feladatot, majd hasonlítsa össze a saját programját az általunk közölt megoldással.
A rutin kezdőcíme legyen $C440:

```
0 REM **** P29. ****
1 :
2 :
00FA              90 AD      =    $FA
00FB              95 AD1     =    $FB
C440              97         *=   $C440
C440   A0 C0     100         LDY  #>$C000
C442   84 FB     102         STY  *AD1
C444   A0 00     104         LDY  #<$E000
C446   84 FA     106         STY  *AD
C448   A2 04     110         LDX  #4
C44A   91 FA     130 SYMB1   STA  (AD),Y
C44C   C8        140         INY
C44D   D0 FB     150         BME  SYMB1
C44F   E6 FB     160         INC  *AD1
C451   CA        170         DEX
C452   D0 F6     180         BNE  SYMB1
C454   60        190         RTS
                 200         .EN
```

Reméljük, hogy az Ön megoldása hasonlít egy kicsit a miénkre!

Az új program logikája hasonló a P27-es program logikájához, attól csak annyiban tértünk el, hogy a 180-as sorban is a SYMB1 címkére ugrunk, hiszen az Y regiszter tartalma ebben a sorban még nulla. A SYMB2 címke feleslegessé vált.
A program kezdőcíme 50240 ($C440).
Ha ki akarjuk próbálni a programot, ne feledkezzünk meg arról, hogy mielőtt az X regiszter tartalmát használnánk, a 780-as címre be kell írni POKE utasítással a színkódot (POKE 780,16).

A fenti előkészületek után térjünk rá a grafikakezelés legfontosabb programjára, amely kivilágítja, ill. törli az egyes képernyőpontokat. A program kidolgozása során meg fogunk ismerkedni a gépi kódú programozás további technikai részleteivel, többek között a logikai műveletek jelentőségével. A pontok ábrázolásához ismernünk kell a képernyőtár és a képernyőpontok közötti hozzárendelési szabályt. A hozzárendelés törvényét szemlélteti a következő táblázat.

	0. oszlop	1-es oszlop	...	39-es oszlop
	0	8		312
	1	9		313
	2	10		314
0. sor	3	11	...	315
	4	12		316
	5	13		317
	6	14		318
	7	15		319
	320	328		632
	321	329		633
	322	330		634
1-es sor	323	331	...	635
	324	332		636
	325	333		637
	326	334		638
	327	335		639
...	...			
	7680	7688		7992
	7681	7689		7993
	7682	7690		7994
24-es sor	7683	7691	...	7995
	7684	7692		7996
	7685	7693		7997
	7686	7694		7998
	7687	7695		7999

A grafikus képernyő is 40 oszlopra és 25 sorra van felbontva. A grafikus tárban minden 8×8-as négyzetnek (karakternek) megfelel 8 byte, úgy, hogy minden byte a négyzet egy pontsorára vonatkozik. Az adott sor nyolc pontja megfelel a byte 8 bitjének, a 7. bit a legszélső bal oldali pontnak, a 6. bit jobbra a következő pontnak stb.:

bitszám	7	6	5	4	3	2	1	0
tartalom	1	0	0	0	1	1	0	0

A fenti byte bitmintája alapján a képernyőn az adott pontsorban, balról az első, ötödik és hatodik pont látható a képernyőn.
A kényelmesebb kezelhetőség érdekében minden ponthoz hozzárendelünk egy vízszintes irányú (X) és egy függőleges irányú (Y) koordinátát. A vízszintes koordináta 0-tól 319-ig, a függőleges koordináta pedig 0-tól 199-ig terjedhet. Keresnünk kell egy olyan algoritmust, amellyel kiszámítjuk, hogy a képernyő adott koordinátájú pontjához a képernyőtár hányadik byte-ja, és azon belül melyik bit tartozik.
A feladat nem egyszerű, alaposan át kell gondolnunk. A 0-tól 7-ig, 8-tól 15-ig, 16-tól 23-ig stb. terjedő X koordinátájú pontokhoz tartozó bit a képernyőtár azonos byte-jában van. Ezért a byte sorszámának meghatározásában az X koordináta alsó 3 bitje nem játszik szerepet. Egy bináris szám alsó három bitjét az AND logikai művelettel törölhetjük. Az AND művelet eredménye csak akkor 1, ha mindkét operandus értéke 1. Az alábbi művelet törli az X alsó három bitjét

X AND %11111000

Próbáljuk ki az utasítást BASIC nyelven:

PRINT X AND 248

Milyen szerepet tölt be az Y koordináta a byte sorszámának meghatározásában? Ha Y értéke 0 és 7 közé esik, a fenti művelet eredményéhez egyszerűen hozzá kell adni, ha 7-nél nagyobb, el kell osztani 8-cal (ugyancsak AND művelettel), a hányadost pedig a maradékot figyelmen kívül hagyva meg kell szorozni 40-nel, és a kapott értéket az előző művelet eredményéhez hozzá kell adni. A számítás során azt is figyelembe kell vennünk, hogy az X koordináta értéke 255-nél nagyobb érték is lehet, tehát ha gépi kódban programozunk, ezt a számot két byte-on kell tárolnunk. Osszuk fel az X koordinátát alsó (XL) és felső (XH) byte-ra. A felső byte értéke mindig 0 vagy 1, hiszen X maximális értéke 319.
Gondolatmenetünk alapján, az (X, Y) koordinátájú ponthoz tartozó byte képernyőtárbeli sorszámát a következő BASIC utasítással határozhatjuk meg:

PRINT XH*256 + (XL AND 248) + (Y AND 7) + 40*(Y AND 248)

A képlet helyes, nincs más dolgunk mint lefordítani gépi kódú nyelvre. A használt regiszterek:

Y ⇒ Y-koordináta
A ⇒ XL-koordináta
X ⇒ XH-koordináta

Az X-koordináta két, az Y-koordináta egy byte-ot foglal el, és a fenti összeg kiszámítását szintén 16 biten kell elvégeznünk.

```
0 REM **** P30. ****
1 :
2 :
100 XL = $FA
110 XH = $FB
```

```
120 SUML = $FC
130 SUMH = $FD
140 *= $C460
150 STA *XL
160 STX *XH ; X KOORDINATA MEGJEGYZESE
170 TYA ; Y KOORDINATA
180 AND #$F8 ;
```

A műveletek elvégzése során figyelembe kell vennünk, hogy a processzor nem tud számokat szorozni egymással. A szorzási műveletet meg kell írnunk. Emlékezzünk vissza arra, hogy ha egy bináris számot egy helyiértékkel balra eltolunk, az eredeti érték megduplázódik. A 40-nel való szorzást vissza kell vezetnünk a szám többszöri megduplázására és az összeadásra:

$$A*40 = A*2*2 + A*2*2*2*2$$

Az összeget ismét megduplázva, az eredmény tényleg az eredeti szám 40-szerese.

```
0 REM **** P31/1. ****
1 :
2 :
190 STA *$FE ; KESOBBI MEGORZESRE
200 STA *SUML
210 LDA #0
220 STA *SUMH ; HI-BYTE TORLES
230 ASL *SUML ; ELTOLAS BALRA
240 ROL *SUMH ; ATVITEL FIGYELEMBEVETELE
250 ASL *SUML
260 ROL *SUMH ; ISMET ELTOLAS
```

A 16 bites eltolást a ROL utasítással kell elvégeznünk. A felső byte (SUMH) 7. bitje az eltolás során az átviteli (carry) biten keresztül a felső byte 0. bitjébe kerül. A kapott értékek összeadása:

```
0 REM **** P31/2. ****
1 :
2 :
270 CLC ; ATVITEL TORLESE
280 LDA *SUML
290 ADC *$FE
300 STA *SUML ; AZ EREDMENY TAROLASA
310 LDA *SUMH
320 ADC #0
330 STA *SUMH
```

Miért kellett a SUMH változóhoz nullát adni? Az első összeadás során fellépő átvitelt ugyanis csak a következő összeadásnál tudjuk figyelembe venni.
Az eredményt ismét meg kell kétszereznünk:

```
0 REM **** P31/3. ****
1 :
2 :
340 ASL *SUML
350 ROL *SUMH
360 ASL *SUML
370 ROL *SUMH ; HAROMSZORI DUPLAZAS
380 ASL *SUML
390 ROL *SUMH
```

A feladat nehezén már túl vagyunk. Adjuk össze a két kifejezés értékét:

```
0 REM **** P31/4. ****
1 :
2 :
400 TYA ;Y KOORDINATA AZ AKKUMULATORBA
410 AND #7
420 CLC
430 ADC *SUML
440 STA *SUML
450 LDA *SUMH
460 ADC #0
470 STA *SUMH
```

Ismét 16 bites összeadást végeztünk, az átvitel figyelembevételével. Az AND logikai művelet:

```
0 REM **** P31/5. ****
1 :
2 :
480 CLC
490 LDA *XA
500 AND #$F8
510 ADC *SUML
520 STA *SUML
530 LDA *XH
540 ADC *SUMH
550 STA *SUMH
```

A grafikus tár nem a 0-s, hanem a $E000-s tárcímen kezdődik. a kapott sorszámhoz ezt az értéket hozzá kell adnunk:

```
0 REM **** P31/6. ****
1 :
2 :
560 CLC
570 LDA #<$E000
580 ADC *SUML
590 STA *SUML
600 LDA #>$E000
610 ADC *SUMH
620 STA *SUMH
```

Végeredményként a SUML/SUMH (alsó, felső byte) változókban megkaptuk a keresett byte sorszámát. Határozzuk meg a byte-on belül a ponthoz tartozó bitet! A bit sorszámát az X-koordináta alsó három bitjéből kapjuk meg, a következő táblázat alapján:

X		Bit
0	⇒	7
1	⇒	6
2	⇒	5
3	⇒	4
4	⇒	3
5	⇒	2
6	⇒	1
7	⇒	0

Az egymáshoz rendelt számok bináris alakjában a bitek páronként ellentétes értékűek. A „kizáró VAGY" logikai művelet erre éppen alkalmas:

```
0 REM **** P31/7. ****
1 :
2 :
630 LDA #XL
640 AND #7
650 EOR #7
```

Készen vagyunk mind a byte, mind pedig a bit byte-on belüli sorszámának meghatározásával.
Hogyan kapjuk meg a keresett bit értékét? Annyi helyiértékkel toljuk balra a kapott byte tartalmát, amennyi a byte-on belül a bit pozíciója:

```
0 REM **** P31/8. ****
1 :
2 :
660 TAX ; A POZICIO BETOLTESE X-BE
670 LDA #1
680 SHIFT DEX
690 BMI OK
700 ASL A ; BIT ELTOLASA BALRA
710 BNE SHIFT
720 OK ...
```

Az eltolásokat az X regiszterben számláljuk. Valahányszor egy eltolást elvégeztünk, az X regiszter tartalmát 1-gyel csökkentjük mindaddig, amíg értéke 0 nem lesz. Eközben a bitpozíciónak megfelelő értéket az akkumulátorban tároljuk, és onnan a kiszámított tárcímre írjuk. Figyelnünk kell arra is, hogy a soron következő bit értéke milyen, hiszen ettől függ, hogy az adott pont látható a képernyőn, vagy sem. A byte eredeti tartalmát tehát meg kell őriznünk, és az újat minden lépésben a VAGY logikai művelettel hozzá kell fűznünk. Végül a módosított byte-ot az eredeti helyén tárolnunk kell. A tárolást indirekt indexelt címzéssel végezzük. A nulláslap címet már előzetesen kiszámítottuk. Az indirekt indexelt címzés az Y regisztert használja, tehát az Y regisztert törölnünk kell.

```
0 REM **** P31/9. ****
1 :
2 :
720 OK LDY #0
730 ORA (SUML),Y
740 STA (SUML),Y
750 RTS
```

Ezzel az adott koordinátájú pontot a képernyőn láthatóvá tettük. Néhány apróságot azonban még át kell gondolnunk:
A 730-as sorban beolvastunk egy értéket a $E000–$FFFF tárterületről. A C 64-es azonban mindig a ROM területről olvas, ha előtte nem közöltük az operációs rendszerrel, hogy a RAM-ból szeretnénk olvasni. Az átkapcsolás helye a processzor-port 1-es címe. Ugyanakkor le kell tiltanunk a megszakítást is, ugyanis átkapcsolás után az interrupt rutinok a lezárt ROM területen vannak.
A program végén az alaphelyzetet vissza kell állítanunk: visszakapcsolunk a ROM-ra, és a megszakítást engedélyezzük:

```
0 REM **** P31/10. ****
1 :
2 :
730 LDX #$34 ; RAM KONFIGURACIO
740 SEI ; AZ INTERRUPT LETILTASA
750 STX *1
760 ORA (SUML),Y
770 STA (SUML),Y ; PONT MEGJELENITES
780 LDX #$37 ; ROM KONFIGURACIO
790 STX *1
800 CLI ; INTERRUPT FELSZABADITASA
810 RTS
820 .EN
```

Nézzük át a program teljes assembler listáját:

```
0 REM **** P32. ****
1 :
2 :
00FA            100 XL       =      $FA
00FB            110 XH       =      $FB
00FC            120 SUML     =      $FC
00FD            130 SUMH     =      $FD
C460            140          *=     $C460
C460 85 FA      150          STA    *XL
C462 86 FB      160          STX    *XH       ; AZ X KOORDINATA TAROLASA
C464 98         170          TYA              ; AZ Y KOORDINATA
C465 29 F8      180          AND    #$F8
C467 85 FE      190          STA    *$FE
C469 85 FC      200          STA    *SUML
C46B A9 00      210          LDA    #0
C46D 85 FD      220          STA    *SUMH
C46F 06 FC      230          ASL    *SUML
C471 26 FD      240          ROL    *SUMH
C473 06 FC      250          ASL    *SUML
C475 26 FD      260          ROL    *SUMH
C477 18         270          CLC              ; AZ ATVITEL TORLESE
C478 A5 FC      280          LDA    *SUML
C47A 65 FE      290          ADC    *$FE
C47C 85 FC      300          STA    *SUML
C47E A5 FD      310          LDA    *SUMH
C480 69 00      320          ADC    #0
C482 85 FD      330          STA    *SUMH
C484 06 FC      340          ASL    *SUML
C486 26 FD      350          ROL    *SUMH
C488 06 FC      360          ASL    *SUML
C48A 26 FD      370          ROL    *SUMH
C48C 06 FC      380          ASL    *SUML
C48E 26 FD      390          ROL    *SUMH
C490 98         400          TYA              ; Y KOORDINATA
C491 29 07      410          AND    #7
C493 18         420          CLC
C494 65 FC      430          ADC    *SUML
C496 85 FC      440          STA    *SUML
C498 A5 FD      450          LDA    *SUMH
C49A 69 00      460          ADC    #0
C49C 85 FD      470          STA    *SUMH
C49E 18         480          CLC
C49F A5 FA      490          LDA    *XL
C4A1 29 F8      500          AND    #$F8
C4A3 65 FC      510          ADC    *SUML
C4A5 85 FC      520          STA    *SUML
C4A7 A5 FB      530          LDA    *XH
C4A9 65 FD      540          ADC    *SUMH
C4AB 85 FD      550          STA    *SUMH
```

```
C4AD 18          560          CLC
C4AE A9 00       570          LDA     #<$E000
C4B0 65 FC       580          ADC     *SUML
C4B2 85 FC       590          STA     *SUML
C4B4 A9 E0       600          LDA     #>$E000
C4B6 65 FD       610          ADC     *SUMH
C4B8 85 FD       620          STA     *SUMH
C4BA A5 FA       630          LDA     *XL
C4BC 29 07       640          AND     #7
C4BE 49 07       650          EOR     #7
C4C0 AA          660          TAX
C4C1 A9 01       670          LDA     #1
C4C3 CA          680 SHIFT    DEX
C4C4 30 03       690          BMI     OK
C4C6 0A          700          ASL     A
C4C7 D0 FA       710          BNE     SHIFT
C4C9 A0 00       720 OK       LDY     #0
C4CB A2 34       730          LDX     #$34
C4CD 78          740          SEI
C4CE 86 01       750          STX     *1
C4D0 11 FC       760          ORA     (SUML),Y
C4D2 91 FC       770          STA     (SUML),Y
C4D4 A2 37       780          LDX     #$37
C4D6 86 01       790          STX     *1
C4D8 58          800          CLI
C4D9 60          810          RTS
                 820          .EN

C460 / C4DA / 007A
A FORRAS FILE: 3.SRC
 0  HIBA

OK      C4C9      SHIFT   C4C3       SUMH    00FD       SUML    00FC
XH      00FB      XL      00FA
```

Készítsünk egy BASIC programot a gépi kódú program működésének ellenőrzésére:

```
0 REM **** P33. ****
1 :
2 :
100 SYS 50176 : REM GRAFIKA BE
110 SYS 50208 : REM GREFIKA TORLESE
120 POKE 780,16 : REM FEKETE/FEHER
130 SYS 50240 : REM SZIN INICIALIZALAS
140 FOR X=0 TO 319
150 POKE 780,X AND 255 : REM X-LO
160 POKE 781,X / 256 : REM X-HI
170 POKE 782,X * 0.625 : REM Y
180 SYS 50272 : REM PONT KIIRASA
190 NEXT
200 GET A$ : IF A$="" THEN 200
210 SYS 50192 : REM KIKAPCSOLAS
```

A program a bal felső saroktól kezdve kirajzol egy átlós irányú egyenest a képernyőre. Egy billentyű leütése után a képernyő visszakapcsol normál üzemmódra.
Most gondoljuk át, hogyan tudnánk egy látható pontot törölni a képernyőről. Világos, hogy a byte és a bit pozícióját ugyanazzal a rutinnal számíthatjuk ki, amit a pont megjelenítéséhez készítettünk. Mindössze annyit kell változtatnunk, hogy a bitet 1 helyett 0-ra kell állítanunk (a 760-as sorban).

Hogyan is működik az ORA művelet?

Az eredeti bitminta	%01001000
A keresett bit	%00010000
Az ORA művelet eredménye	%01011000

A VAGY művelet a keresett bitet 1-re állította. Most ennek az ellenkezőjét kell tennünk: a bitet törölnünk kell. Alkalmazzuk az AND műveletet:

Az eredeti bitminta	%01011000
A törlendő bit	%00010000
Az AND művelet eredménye	%00010000

Furcsa dolog történt! Minden bit törlődött, kivéve azt az egyet, amelyet törölni akartunk! Az AND művelet operandusában minden bit értékének 1-nek kell lennie, kivéve a törlendő bitet. Alkalmazzuk az operandusra az EOR műveletet:

A törlendő pont	%00010000
Az EOR művelet eredménye	%11111111
Az új bitminta	%11101111

A fenti módosítás után végezzük el az AND műveletet:

Az eredeti bitminta	%01011000
Az új bitminta	%11101111
A kívánt bitek törlődtek	%01001000

Építsük be a gépi kódú programba a törlési funkciót. A C kapcsolóval vezéreljük, hogy a pontot a képernyőn törölni kell, vagy megjeleníteni. Ha a C kapcsoló értéke 0, akkor töröljük, ha 1, akkor világítsuk ki a pontot. A C kapcsoló eredeti értékét meg kell őriznünk. Helyezzük el az állapotregiszter tartalmát PHP utasítással átmenetileg a veremben (145-ös sor). A munka végeztével (a 735-ös sorban) töltsük vissza az állapotregiszter tartalmát.

```
0 REM **** P34. ****
1 :
2 :
760 BCC TORL.
770 ORA (SUML),Y
780 BCS OK2
790 TORL. EOR #$FF ; MEGFORDITAS
800 AND (SUML),Y
810 OK2 STA (SUML)
820 LDX #$37
830 STX *1
840 CLI
850 RTS
860 .EN
```

Ha a C kapcsoló tartalma 0, ugrunk a 790-es sorra, ahol EOR $FF művelettel a biteket ellenkezőjére változtatjuk, majd végrehajtjuk az AND-et, végül az eredményt tároljuk. Ha a C kapcsoló értéke 1, ORA műveletet végzünk, majd ugrunk ismét a tárolásra. Az elágazást a BCS utasítással vezéreljük, mivel az ORA művelet nem módosítja a C kapcsoló értékét.

```
0 REM **** P35. ****
1 :
2 :
00FA              100 XL       =     $FA
00FB              110 XH       =     $FB
00FC              120 SUML     =     $FC
00FD              130 SUMH     =     $FD
C460              140          *=    $C460
C460 08           145          PHP
C461 85 FA        150          STA   *XL
C463 86 FB        160          STX   *XH    ; AZ X KOORDINATA TAROLASA
C465 98           170          TYA          ; AZ Y KOORDINATA
C466 29 F8        180          AND   #$F8
C468 85 FE        190          STA   *$FE
C46A 85 FC        200          STA   *SUML
C46C A9 00        210          LDA   #0
C46E 85 FD        220          STA   *SUMH
C470 06 FC        230          ASL   *SUML
C472 26 FD        240          ROL   *SUMH
C474 06 FC        250          ASL   *SUML
C476 26 FD        260          ROL   *SUMH
C478 18           270          CLC          ; AZ ATVITEL TORLESE
C479 A5 FC        280          LDA   *SUML
C47B 65 FE        290          ADC   *$FE
C47D 85 FC        300          STA   *SUML
C47F A5 FD        310          LDA   *SUMH
C481 69 00        320          ADC   #0
C483 85 FD        330          STA   *SUMH
C485 06 FC        340          ASL   *SUML
C487 26 FD        350          ROL   *SUMH
C489 06 FC        360          ASL   *SUML
C48B 26 FD        370          ROL   *SUMH
C48D 06 FC        380          ASL   *SUML
C48F 26 FD        390          ROL   *SUMH
C491 98           400          TYA          ; Y KOORDINATA
C492 29 07        410          AND   #7
C494 18           420          CLC
C495 65 FC        430          ADC   *SUML
C497 85 FC        440          STA   *SUML
C499 A5 FD        450          LDA   *SUMH
C49B 69 00        460          ADC   #0
C49D 85 FD        470          STA   *SUMH
C49F 18           480          CLC
C4A0 A5 FA        490          LDA   *XL
C4A2 29 F8        500          AND   #$F8
C4A4 65 FC        510          ADC   *SUML
C4A6 85 FC        520          STA   *SUML
C4A8 A5 FB        530          LDA   *XH
C4AA 65 FD        540          ADC   *SUMH
C4AC 85 FD        550          STA   *SUMH
C4AE 18           560          CLC
C4AF A9 00        570          LDA   #<$E000
C4B1 65 FC        580          ADC   *SUML
C4B3 85 FC        590          STA   *SUML
C4B5 A9 E0        600          LDA   #>$E000
C4B7 65 FD        610          ADC   *SUMH
C4B9 85 FD        620          STA   *SUMH
C4BB A5 FA        630          LDA   *XL
C4BD 29 07        640          AND   #7
C4BF 49 07        650          EOR   #7
C4C1 AA           660          TAX
C4C2 A9 01        670          LDA   #1
C4C4 CA           680 SHIFT    DEX
C4C5 30 03        690          BMI   OK
C4C7 0A           700          ASL   A
C4C8 D0 FA        710          BNE   SHIFT
C4CA A0 00        720 OK       LDY   #0
```

```
C4CC A2 34        730          LDX    #$34
C4CE 28           735          PLP
C4CF 78           740          SEI
C4D0 86 01        750          STX    #1
C4D2 90 04        760          BCC    TORLES
C4D4 11 FC        770          STA    (SUML),Y
C4D6 B0 04        780          BCS    OK2
C4D8 49 FF        790  TORLES  EOR    #$FF
C4DA 31 FC        800          AND    (SUML),Y
C4DC 91 FC        810  OK2     STA    (SUML),Y
C4DE A2 37        820          LDX    #$37
C4E0 86 01        830          STX    #1
C4E2 58           840          CLI
C4E3 60           850          RTS
                  860          .EN

C460 / C4E4 / 0084
A FORRAS FILE: 4.SRC
 0  HIBA

TORLES C4D8     OK     C4DC     OK2    C4DC     SHIFT   C4C4
SUMH   00FD     SUML   00FC     XH     00FB     XL      00FA
```

Módosítsuk egy kicsit a BASIC mintaprogramot is:

```
0 REM **** P36. ****
1 :
2 :
100 SYS 50176 : REM GRAFIKA BE
110 SYS 50208 : REM GREFIKA TORLESE
120 POKE 780,16
130 SYS 50240 : REM SZINBEALLITAS
140 I=1
150 FOR X=0 TO 319
160 POKE 780,X AND 255 : REM X-LO
170 POKE 781,X / 256 : REM X-HI
180 POKE 782,X * 0.625 : REM Y
190 POKE 783,I : REM IRAS/TORLES
200 SYS 50272 : NEXT
210 GET A$ : IF A$="" THEN I=1-I:GOTO 150
220 SYS 50192 : REM GRAFIKA KI
```

Ez a program megrajzolja, majd törli az átlós egyenest. Azt, hogy a pontokat megjeleníteni vagy törölni akarjuk, a 783-as tárcímen adjuk meg, az állapotregiszter tartalmának beállításával. A C kapcsolónak megfelelő bit értékét kívánság szerint 1-re vagy 0-ra állítjuk.
A program futását egy billentyű leütésével megszakíthatjuk. Megszakítás után a képernyő eredeti tartalma változatlanul megmarad. Ha a későbbiek során vissza akarunk kapcsolni grafikus képernyőre, és szükségünk van az előző képre, hagyjuk ki a programból a képernyő törlését. Próbáljuk ki a programmal a képernyő lehetséges színkombinációit is!

```
0 REM **** P37. ****
1 :
2 :
100 SYS 50176 : REM GRAFIKA BE
110 SYS 50208 : REM GREFIKA TORLESE
120 POKE 780,16
130 SYS 50240 : REM SZINBEALLITAS
140 REM
150 FOR X=70 TO 150 : FOR Y=X TO 199
160 POKE 780,X : REM X-LO
170 POKE 781,0 : REM X-HI
```

```
180 POKE 782,Y : REM Y
190 POKE 783,1 : REM BEALLITAS
200 SYS 50272 : NEXT : NEXT
210 FOR C=0 TO 255
220 FOR I=1 TO 500 : NEXT
230 POKE 780,C
240 SYS 50240 : REM SZIN
250 NEXT
260 SYS 50192 : REM GRAFIKA KI
```

A fenti program kirajzol a képernyőre egy figurát, majd ezt a figurát a képernyő összes lehetséges színkombinációjában megjeleníti.

Ebben a gépi kódú programban közelebbről megismerkedhettünk az indexelt címzésmóddal és a logikai műveletekkel. Felhasználtuk a vermet adatok átmeneti tárolására. Megismerkedhettünk a 16 bites összeadással és eltolással. Azonban a gépi kódú programozás egyik legfontosabb technikai lehetőségét, a szubrutinok szerkesztését egyenlőre nem érintettük, erre szeretnénk kitérni a következő fejezetben.
Az eddigi ismeretekkel, ennyi gyakorlattal az Olvasó nyugodtan nekivághat egy hardcopy program megírásának. Nyomtassuk ki a grafikus képernyő tartalmát! A program elkészítéséhez ismernünk kell a nyomtató működését. A grafikus jeleket a nyomtatók általában oszlopokra bontva értelmezik úgy, hogy egy átvitt byte tartalma nyolc egymás fölötti pontra vonatkozik. Sajnos az egyes nyomtatótípusok általában nem azonos módszerrel dolgoznak. Ha például a nyomtatónk oszlopos kijelzéssel dolgozik, az előző programbeli számításokat nem tudjuk felhasználni a nyomtatás során, nem tudunk egyszerűen byte-ról byte-ra nyomtatni, hiszen a képernyőtár felépítése a pontsorok vízszintes tárolásán alapszik. Ebben az esetben 8 egymást követő byte tartalmát bitenként szét kell választani, majd a nyomtatónak megfelelően logikai műveletekkel újra fel kell építeni. Ha mindez túlságosan bonyolultnak tűnik, rajzoljuk meg programozás előtt a feladat folyamatábráját. Az elkészült programot teszteljük az egylépéses szimulátorprogrammal, amelyet az előző fejezetben ismertettünk.

8. BASIC BŐVÍTÉSEK ÉS AZ OPERÁCIÓS RENDSZER RUTINJAINAK FELHASZNÁLÁSA

A grafikát kezelő gépi kódú programok kapcsán megismerkedtünk a paramétercsere egyik módszerével, nevezetesen a regiszterek tartalmát POKE utasításokkal határoztuk meg. Most megismerkedünk egy elegánsabb, mégis könnyen programozható eljárással.
A paramétereket ugyanúgy fogjuk átadni, ahogyan azt a BASIC interpreter csinálja. Az eljárás bemutatásához nézzünk először egy egyszerű POKE utasítást:

```
POKE A, B
```

Az Olvasó biztosan tudja, hogy az A és B változók helyett a POKE utasításba tetszőleges kifejezéseket is írhatunk, pl.:

```
POKE A(1000)/750*INT(X%/9), EXP (ABS(SIN(3*A)))+2
```

Az ilyen összetett kifejezéseket a BASIC interpreter egy általános rutinja átveszi, és kiértékeli. Hasznosítsuk ezt a rutint saját céljainkra! A rutinnak vannak olyan beugrási pontjai, amelyek ráadásul bizonyos ellenőrzéseket is elvégeznek. Megvizsgáljuk például, hogy a POKE utasítás első paramétere a megengedett számtartományba, azaz 0 és 65535 közé esik-e.
Ha nem, a rutin kiírja az ILLEGAL QUANTITY hibaüzenetet. Hasonló vizsgálatot végez a második paraméterre is, ahol a megengedett számtartomány a 0 és 255 közé eső számok halmaza.
Hogyan tudnánk ezeket a rutinokat hasznosítani? Ehhez meg kell először ismerkednünk a szubrutinok programozástechnikájával.
A szubrutinokkal biztosan találkozott már az Olvasó a BASIC nyelvben. A szubrutinok szervezésének két alapvető utasítása a BASIC nyelvben a GOSUB és a RETURN.
A GOSUB elágazó utasítás, mellyel a program tetszőleges sorára ugorhatunk. A GOTO utasítással is szervezhetünk programbeli elágaztatást, azonban a GOTO utasítással szemben a GOSUB utasítással vezérelt elágazásoknál az interpreter „megjegyzi" azt a programsort, amelyet az elágazás előtt végrehajtott. A szubrutin végrehajtása után, a RETURN utasítás hatására az interpreter visszaadja a vezérlést a GOSUB utasítást követő BASIC sorra, hiszen előzetesen tárolta a visszatérési címet, így a program futása ott folytatódik, ahol a rutin hívása előtt megszakadt. A fenti BASIC utasításoknak megfelelő utasítások a 6510-es processzor gépi kódjában a:

JSR és az RTS

A JSR (Jump to SubRoutine) vezérli az elágazást, az RTS (ReTurn from Subroutine) utasítás pedig szervezi a visszatérést a szubrutinból.
A programokat a BASIC nyelvhez hasonlóan, a gépi kódban is akkor célszerű szubrutinokból felépíteni, ha egy-egy részfeladatot többször végre kell hajtanunk. Ilyenkor célszerűtlen lenne ugyanazokat a programrészleteket többször megírni, fáradságot és helyet takaríthatunk meg, ha a részfeladatokra szubrutinokat készítünk. Az alprogramokat úgy kell megírni, hogy lehetőleg általánosak legyenek. Célszerű például ügyelni arra, hogy az alprogramban csak olyan konstansok szerepeljenek, amelyek minden hívásnál azonosak. Hogyan dolgozza fel a processzor a JSR utasítást? Mielőtt átadná a vezérlést a megadott sorra, tárolja az éppen végrehajtott utasítás címét (a programszámláló értékét). A tárolás helye a verem, amely a

$100–$1FF tárterületen található. A szubrutin végét az RTS utasítás jelzi. Hatására a processzor elővesz a veremből az ott utoljára elhelyezett két byte-ot, és ezeket visszatérési címként értelmezi. A program futása során a verem tartalma állandóan változik, és a processzor egy speciális regiszter, a veremmutató (SP) értékéből tudja, hogy az utoljára bekerült érték a veremben hol található.
Nézzünk erre egy konkrét példát:

```
C000  20  00  C1  JSR  $C100
C003...

C100  60          RTS
```

Amikor a program végrehajtása a $C000-s címhez ér, a processzor átadja a vezérlést a $C1000-s sorra. Ebben a sorban RTS utasítást talál, így azonnal visszatér a szubrutinból a hívást követő programsorra, azaz a fenti példában a $C003-as címre.
Nézzük meg, hogyan változik a program végrehajtása közben a verem tartalma, és a veremmutató értéke:

Cím	Utasítás	Veremmutató	Verem	
$C000	JSR $C100	$F9	$01F9	XX

Tegyük fel, hogy a veremmutató értéke $F9. A $01F9-es címen az előző művelet adatai találhatók. Amikor a processzor a JSR utasítás végrehajtásához ér, a programszámláló értékét megnöveli 2-vel, majd a kapott címet felbontja alsó és felső byte-ra. A felső byte-ot elhelyezi a veremmutató által megadott címen, a veremmutató értéket pedig 1-gyel csökkenti:

Cím	Utasítás	Veremmutató	Verem	
$C000	JSR $C100	$F8	$01F9	C0
			$01F8	XX

Most elhelyezi a veremben a visszaugrási cím alsó byte-ját, és a veremmutató értékét ismét csökkenti 1-gyel:

Cím	Utasítás	Veremmutató	Verem	
$C100	RTS	$F7	$01F9	C0
			$01F8	02
			$01F7	XX

A teljes cím bekerült a verembe, és a veremmutató értéke 2-vel csökkent. A veremmutató ismét a verem első címe.
A szubrutinból való visszatérés során a processzor a fentieket fordított sorrendben hajtja végre. Megnöveli 1-gyel a veremmutatót, és a kapott címen található értékét a visszaugrási cím alsó byte-jának tekinti. A veremmutatót ismét megnöveli, és a megfelelő byte-ot ismét elővesz a veremből.

Cím	Utasítás	Veremmutató	Verem	
$C100	RTS	$F9	$01F9	C0
			$01F8	02
			$01F7	XX

A $02 és $C0 értékekből előállítja a $C002 címet, és ezt a programszámláló értékének tekinti. Végül megnöveli a programszámláló értékét 1-gyel, és a kapott címről betölti a következő végrehajtandó utasítást.

Cím	Utasítás	Veremmutató	Verem
$C003	...	$F9	$01F9 C0
			$01F8 02
			$01F7 XX

A veremmutató értéke ugyanaz, mint ami programhívás előtt volt. Ezzel a technikával több szubrutint egymásba skatulyázhatunk. Ha a szubrutinból egy másik szubrutint hívnánk, a processzor a visszatérési címet a veremben a $F5-ös címre helyezné el. Az RTS utasítás hatására ezt a címet olvasná vissza, és a veremmutatót $F7-re állítaná. Minden RTS után az utoljára elhelyezett visszatérési cím alapján képezi a végrehajtandó utasítás címét.
Természetesen a programozónak nem kell foglalkoznia a verem és a veremmutató tartalmával, a fentieket a processzor minden szubrutin hívásakor automatikusan elvégzi.
A verem nemcsak a visszatérési címek, hanem az akkumulátor, ill. az állapotregiszter tartalmának átmeneti tárolására is szolgál. Az akkumulátor tartalmát a PHA és a PLA utasítás, az állapotregiszter tartalmát pedig a PHP és a PLP utasítás tárolja, ill. tölti vissza a verembe. A veremmutató értéke tároláskor 1-gyel csökken, visszatöltésnél 1-gyel nő. Közvetve bármely regiszter tartalmát a verembe írhatjuk az akkumulátoron keresztül:

```
PHA ;   Az akku, az
TYA ;
PHA ;   Y regiszter tartalma
TXA
PHA ;   ill. az X regiszter tartalma a verembe
...
PLA ;
TAX ;   Az X regiszter tartalmának, az
PLA ;
TAY ;   Y regiszter tartalmának
PLA ;   és az akku tartalmának visszatöltése a veremből
```

A regiszterek tartalmának átmeneti tárolása közben mindig ügyeljünk arra, hogy a visszatöltés sorrendje ellentétes legyen a tárolás sorrendjével. A veremből először mindig az utoljára tárolt értéket kell visszaolvasni! (LIFO).
A szubrutinok hívása során gyakran szükséges, hogy a regiszterek tartalmát megőrizzük, hiszen előfordulhat, hogy tartalmuk a szubrutin végrehajtása közben megváltozik.
A verem működését nagyon jól szemlélteti az egylépéses szimulátor, hiszen minden lépésben látható a regiszterek tartalma.

A fenti kitérő után térjünk vissza a paraméterek átadásának problémájára.
A BASIC interpreter tartalmaz egy GETBYT nevű rutint, amely beolvas egy kifejezést a BASIC szövegből és megvizsgálja, hogy annak értéke 0 és 255 közé esik vagy sem, majd a kifejezés értékét az X regiszterbe helyezi. A rutin kezdőcíme: $B79E. További két rutin egy 16 bites (0-tól 65535-ig terjedő értékű) számot olvas be a BASIC területről.
Az FRMNUM rutin beolvassa, a GETADR rutin pedig kiértékeli a kifejezést. Ha a kifejezés

értéke 0 és 65535 közé esik, átadja a $14 (alsó byte) és $15 (felső byte) értékeket. A rutinok címét az alábbiakban közöljük.
Több paraméter esetén az egyes paramétereket el kell választanunk egymástól. Az elválasztó karakter a BASIC nyelvben lehet a vessző. A BASIC interpreter CKHCOM rutinja megvizsgálja, hogy a BASIC szöveg következő karaktere vessző vagy sem. Az interpreter CHRGOT és CHRGET rutinjai beolvasnak egy sort a BASIC programból. A CHRGOT betölti az akkumulátorba azt a karaktert, amelyre a programszámláló mutat, a CHRGET pedig megnöveli eggyel a mutató értékét. Ugyanakkor a betöltött karaktertől függően a rutinok a megfelelő kapcsolókat is módosítják. Például ha a karakter nullabyte (a BASIC sor vége), vagy kettőspont, a Z kapcsoló érték 1 lesz. Ha számjegyet olvastunk, a C kapcsoló 0-ra vált. Az említett rutinok tárcímei.

```
GETBYT   $B79E
FRMNUM   $AD8A
GETADR   $B7F7
CHKCOM   $AEFD
CHRGOT   $0079
CHRGET   $0073
```

Ha egymást követően egy 16 bites és egy 8 bites értéket olvasunk (mint pl. a POKE utasításnál), a GETPAR rutint kell használnunk, amely megkeresi az elválasztó vesszőt. A GETPAR rutinból az interpreter rendre meghívja a FRMNUM, GETADR, CHKCOM és GETBYT rutinokat.

```
GETPAR   $B7EB
```

Ezeket a rutinokat felhasználhattuk volna a grafikus programban, hiszen az X koordináta éppen egy 16 bites (0-tól 319-ig), az Y koordináta pedig egy 8 bites (0-tól 199-ig) számérték. Túllépve a 65535, ill. a 255 értékeket, az interpreter automatikusan az ILLEGAL QUANTITY hibaüzenetet küldi. Az interpreter ellenőrzését még megtoldhatjuk egy saját ellenőrző eljárással, amelyben a feladattól függő értékhatárokat vizsgáljuk, és hibás adatok esetén ismét a hibaüzenetet kijelző rutinra ugrunk, amely a $B248-as címen található.
A rendszerrutinokra épített paraméterátadás esetén a gépi kódú rutin hívására a

```
SYS 50240, X,Y
```

utasítást használhatjuk. Ezzel megtakarítottuk a POKE utasításokat, és a rutinok hívása is áttekinthetőbb lett. Írjuk át a pontokat ábrázoló gépi kódú programot az elmondottak szerint.

```
0 REM **** P38/1. ****
1 :
2 :
100 JSR CHKCOM ; VESSZO KOVETKEZIK ?
110 JSR GETTAR ; KOORDINATAK BETOLTESE
120 STX YKOOR ; Y KOORDINATA FIGYELES
130 LDA $14
140 STA XL ; X KOORDINATA LO
150 LDA $15
160 STA XH
```

Először megkeressük a SYS utasításban az elválasztó vesszőt, majd meghívjuk a paramétereket betöltő rutinokat. Az Y koordináta először az X regiszterbe, majd az YKOOR tárcímre kerül. A 16 bites X koordinátát pedig a $14/$15-ös tárcímekről az XL és XH változókba töltjük. A paraméterek tárolását követi az értékhatárok ellenőrzése. Az Y koordinátát összehasonlítjuk 200-zal. Ha a paraméter kisebb mint 200, a programfutás folytatódik, egyébként ugrunk a hibaüzenetet kijelző rutinra. Az X koordináta vizsgálatát 16 bites összehasonlítással kell elvégeznünk.

```
0 REM **** P38/2. ****
1 :
2 :
170 CPX #200 ; Y >= 200 ?
180 BCC OK
190 HIBAJELZES
200 OK LDA XH
210 CMP #>320
220 BCC OK1
230 BNE HIBA
240 LDA XA
250 CMP #<320
260 BCS HIBA
270 OK1 ...
```

Ha az X regiszter tartalma, az Y koordináta kisebb 200-nál, a C kapcsoló törlődik, és elágazunk az OK szimbolikus címre, egyébként a C kapcsoló értéke 1 lesz, és a HIBA címre ugrunk ($B248). Ha az Y koordináta értéke megfelelő volt, a program futása az X koordináta vizsgálatával folytatódik. Betöltjük az akkumulátorba a felső byte-ot, és összehasonlítjuk 320 felső byte-jával. Ha a kisebb reláció teljesül, akkor az X koordináta értéke is kisebb 320-nál, így az OK1 címre ugrunk. Egyébként a HIBA címen folytatódik a program futása. Ha a felső byte értéke egyenlő 320 felső byte-jával, akkor a koordináta már biztosan 256 és 511 közé esik, folytathatjuk a vizsgálatot az alsó byte-ok összehasonlításával. Betöltjük az alsó byte-ot az akkumulátorba és összehasonlítjuk a 320 alsó byte-jával. Ha a nagyobb reláció teljesül, a hibaüzenetre ugrunk. Egyébként folytatódhat a koordináták feldolgozása.

9. AZ INPUT/OUTPUT MŰVELETEK PROGRAMOZÁSA

A gépi kódú programozás ismertetése során eddig még nem esett szó az alapgép és a külső egységek közötti információáramlásról. Nem beszéltünk még arról, hogy hogyan lehet egy karaktert a billentyűzetről beolvasni, a képernyőre kiírni stb. Nézzük először a BASIC nyelv I/O utasításait.

OPEN
CMD
PRINT#
INPUT#
CLOSE

A gépi kódú nyelv I/O utasításai nagyon hasonlítanak a felsorolt utasításokra. A fenti utasítások mindegyikéhez van az operációs rendszerben egy-egy gépi kódú rutin. Ha ismerjük a rutinok ugrási címeit, programjainkban hivatkozhatunk rájuk. Vegyük sorra ezeket a rendszerrutinokat:

OPEN

Az OPEN rutin három paramétert használ: a logikai file-számot, az egységszámot és a másodlagos címet. Ezeket a paramétereket a SETFLS és a SETNAM rutinok készítik elő. A paramétereket nem magában az OPEN utasításban kell megadni, hanem az OPEN utasítás előtt, az említett rutinok meghívásával.

SETFLS

Hívás előtt be kell tölteni az akkumulátorba a logikai file-számot, az X regiszterbe az egységszámot és az Y regiszterbe a másodlagos címet.

SETNAM

Ezzel a rutinnal meghatározhatjuk a file nevét. A file-név hosszát (ha értéke 0, nincs megadva file-név), és a kezdőcímet (azt a címet, ahol a tárban a nevet elhelyeztük) be kell tölteni az akkumulátorba, ill. az X- (alsó byte) és Y regiszterbe (felső byte).

PRINT

A PRINT rutin működése igen egyszerű. Az akkumulátorba előzetesen betöltött karaktert kiírja a megadott külső egységre. Ha a PRINT hívása előtt OPEN-nel adtunk meg logikai file-t, a kiírás a képernyőre vonatkozik.

CHKOUT

Ez a rutin megfelel a BASIC CMD utasításnak. Ha egy karaktert egy megnyitott file-ba akarunk beírni (és nem a képernyőre), be kell töltenünk az X regiszterbe a file logikai számát, és meg kell hívni a CHKOUT rutint. Ezt követően minden PRINT utasítás a kijelölt külső egységre irányul, mindaddig, amíg az alábbi rutinnal az alapértelmezést vissza nem állítjuk.

CLRCH

Ez a rutin megszünteti a CHKOUT rutin hatását. Paramétert nem használ.

INPUT

Az INPUT rutin beolvas egy karaktert a billentyűzetről és betölti az akkumulátorba. Ha nem a billentyűzetről, hanem egy file-ból akarunk olvasni, a file-t meg kell nyitnunk, és a CHKIN rutinnal meg kell jelölnünk, mint input file-t.

CHKIN

A rutin hívása előtt be kell tölteni az X regiszterbe az input file logikai számát. A hívás eredményeként a rendszer a továbbiakban nem a billentyűzetről, hanem a logikai számmal megjelölt file-ból olvassa az adatokat. A hozzárendelést a CLRCH rutinnal lehet megszüntetni.

CLOSE

A CLOSE utasítás lezárja azt a file-t, melynek logikai száma az akkumulátorban van.

A bemutatott rutinok ugrási címei:

Rutin	Cím
OPEN	$FFC0
SETFLS	$FFBA
SETNAM	$FFBD
PRINT	$FFD2
CHKOUT	$FFC9
CLRCH	$FFCC
INPUT	$FFCC
CHKIN	$FFC6
CLOSE	$FFC3

Nézzünk egy példát: írjuk meg az alábbi BASIC programot gépi kódban:

```
OPEN 1,8,15
PRINT#1, "I"
CLOSE 1
```

A gépi kódú program:

```
0 REM **** P39. ****
1 :
2 :
100 LDA #1 ; LOGIKAI FILESZAM
110 LDX #8 ; EGYSEGSZAM
120 LDY #15 ; MASODLAGOS CIM
130 JSR SETFLS
140 LDA #0
150 JSR SETNAM ; NINCS NEV
160 JSR OPEN ; FILE NYITAS
170 LDX #1 ; LOGIKAI FILESZAM
180 JSR CHKOUT ; KIIRAS A FILE-BA
190 LDA #73 ; "I"
200 JSR PRINT
210 JSR CLRCH
220 LDA #1 ; LOGIKAI FILESZAM
230 JSR CLOSE
240 RTS
```

A 100-tól 120-ig terjedő sorokban betöltjük az OPEN rutin paramétereit és elhelyezzük a megfelelő tárcímre. A 140-es sorban a file nevének hossza helyett 0-t tárolunk, ezzel jelezve, hogy nem használunk file-nevet. A 160-as sorban ugrunk az OPEN rutinra. A PRINT utasítás az 1-es logikai file-ra ír, így az I betű ASCII kódját a lemezegységhez küldjük. Végül a CLRCH rutinnal az alapértelmezést visszaállítjuk, a kiírások ismét a képernyőre vonatkoznak. A 220-as és 230-as sorokban a file-t lezárjuk, majd az RTS utasítással visszatérünk a BASIC interpreterhez.

A következő példában leolvassuk a hibacsatornát és a hibaüzenetet kiírjuk a képernyőre.

```
0 REM **** P40. ****
1 :
2 :
100 OPEN 1,8,15
110 INPUT#1,A,B$,C,D
120 PRINTA;B$;C;D
130 CLOSE 1
```

Mivel a hibaüzenet szövegét közvetlenül a képernyőre írjuk, tárolására nem kell külön változót használnunk. A szöveget addig olvassuk, amíg a statusz (ST) értéke 64 nem lesz, ez jelzi ugyanis a szöveg végét.

Először írjuk meg a programot BASIC nyelven:

```
0 REM **** P41. ****
1 :
2 :
100 OPEN 1,8,15
110 GET#1,A$ : PRINT A$;
120 IF ST <> 64 THEN 110
130 CLOSE 1
```

A gépi kódú program elkészítéséhez tudnunk kell, hogy a státuszváltozó a tár 144-es ($90-es) címén található.

```
0 REM **** P42. ****
1 :
2 :
10 OPEN   = $FFC0
20 SETFLS = $FFBA
30 SETNAM = $FFBD
40 PRINT  = $FFD2
50 CLRCH  = $FFCC
60 INPUT  = $FFCF
70 CHKIN  = $FFC6
80 CLOSE  = $FFC3
90 STATUS = $90 ; STATUSZVALTOZO
100 LDA #1 ; LOGIKAI FILESZAM
110 LDX #8 ; EGYSEGSZAM
120 LDY #15 ; MASODLAGOS CIM
130 JSR SETFLS
140 LDA #0
150 JSR SETNAM ; NINCS NEV
160 JSR OPEN ; FILE NYITAS
170 LDX #1 ; LOGIKAI FILESZAM
180 JSR CHKIN ; BEOLVASAS A HIBACSATORNAROL
190 L1 JSR INPUT ; A JEL BETOLTESE
200 JSR PRINT ; ES KIIRASA
210 BIT STATUS ; STATUSZVIZSGALAT
220 BVC L1
230 JSR CLRCH ; VISSZAALLAS ALAPERTELMEZESRE
240 LDA #1
250 JSR CLOSE
260 RTS
270 .EN
```

A file megnyitására beépítettük az előző rutint a programba. A hibaüzenetet a hibacsatornán keresztül olvassuk le, ezért a 170-es és 180-as sorokban a hibacsatornát jelöltük meg input file-ként. Az X regiszterbe betöltöttük az 1 file-számot, majd meghívtuk a CHKIN rutint. A következő sorok tartalmazzák a tényleges beolvasást. A 190-es sorban beolvasunk egy karaktert a lemezről, majd JSR PRINT utasítással kiírjuk a képernyőre (200-as sor). Olvasás után megvizsgáljuk a státuszváltozó értékét BIT utasítással. Ha a státuszváltozó értéke 64, azaz kettő hatodik hatványa, binárisan: %01000000, akkor elértük a szöveg végét.
Ez esetben a 144-es tárcím hatodik bitje 1. A BIT utasítás átmásolja a hatodik bit értékét a V, a hetedik bit értékét pedig az N kapcsolóba. A szöveg végét jelző elágazást tehát a V kapcsoló értéke alapján kell megszervezni. A BVC utasítás elágazik, ha a V kapcsoló értéke 0, azaz ha esetünkben a státuszváltozó értéke nem egyenlő 64-gyel, tehát ekkor az olvasást folytatnunk kell. Ha a V kapcsoló értéke 1, a JSR CLRCH utasítással visszaállítjuk az alaphelyzetet és lezárjuk a file-t. A program megírása közben ügyeljünk arra, hogy öt karakternél hosszabb szimbólumokkal nem dolgozhatunk.

```
0 REM **** P43. ****
1 :
2 :
5 :    C000                 .OPT P1,00
10 :   FFC0        OPEN      =     $FFC0
20 :   FFBA        SETFLS    =     $FFBA
30 :   FFBD        SETNAM    =     $FFBD
40 :   FFD2        PRINT     =     $FFD2
50 :   FFCC        CLRCH     =     $FFCC
60 :   FFCF        INPUT     =     $FFCF
```

```
70 :  FFC6            CHKIN    =   $FFC6
80 :  FFC3            CLOSE    =   $FFC3
90 :  0090            STATUS   =   $90
100 : C000 A9 01            LDA   #1       ;LOGIKAI FILESZAM
110 : C002 A2 08            LDX   #8       ;KESZULEKSZAM
120 : C004 A0 0F            LDY   #15      ;MASODLAGOS CIM
130 : C006 20 BA FF         JSR   SETFLS
140 : C009 A9 00            LDA   #0       ;NINCS FILENEV
150 : C00B 20 BD FF         JSR   SETNAM
160 : C00E 20 C0 FF         JSR   OPEN     ;FILE NYITAS
170 : C011 A2 01            LDX   #1       ;BEOLVASAS
180 : C013 20 C6 FF         JSR   CHKIN    ;A HIBACSATORNAROL
190 : C016 20 CF FF L1      JSR   INPUT    ;JEL A FLOPPY-ROL
200 : C019 20 D2 FF         JSR   PRINT    ;A JEL KIIRASA
210 : C01C 24 90            BIT   STATUS   ;STATUSZVIZSGALAT
220 : C01E 50 F6            BVC   L1
230 : C020 20 CC FF         JSR   CLRCH
240 : C023 A9 01            LDA   #1
250 : C025 20 C3 FF         JSR   CLOSE    ;A FILE LEZARASA
260 : C028 60               RTS
```

A P 43-as programot a PROFI ASS 64 2.0 programmal fordítottuk le (assembláltuk). Ezt az assemblert a könyv 11. fejezetében fogjuk ismertetni. A 6-os sorban található utasítás hatására a programot a korábban megnyitott logikai file-ba listázhatjuk, míg a gépi kódú program (objectcode) közvetlenül a tárba kerül.

Próbáljuk ki az elkészített programot a

SYS 49152

utasítással. A képernyőn megjelenik a lemezegység hibaüzenete, pl.

00, OK, 00, 00

10. BASIC BETÖLTŐPROGRAMOK KÉSZÍTÉSE

Ha nem rendelkezünk assembler programmal, gondot okozhat, a gépi kódú program olyan tárolása, hogy az bármikor betölthető legyen. A következő módszert javasoljuk:
A teljes gépi kódú programot helyezzük el decimális kódokkal DATA sorokban, majd egy BASIC ciklusban, POKE utasításokkal írjuk be a kódokat a megfelelő tárcímekre. A decimális kódok kiszámítása eléggé fáradságos munka, ezért írtunk egy olyan BASIC programot, ami megírja helyettünk a BASIC betöltőprogramot. Az eljárás a következő: töltsük be először a tárba a gépi kódú programot, majd a P 44-es programot. Indítás után meg kell adnunk a gépi kódú program tárbeli kezdő- és végcímét. A többit már elvégzi helyettünk a program – kiírja nyomtatóra a BASIC betöltőprogram teljes listáját, úgy, hogy közben a DATA sorokhoz kontrollösszeget is képez. A betöltőprogram képezi a DATA sorokban szereplő kódok összegét; és ellenőrzi, hogy az azonos-e az előző programban (P 44-es) kiszámított értékekkel. Az eredmény alapján üzenetet küld a programozónak, figyelmeztetve az esetleges gépelési hibákra. A gyakorlat azt mutatja, hogy a DATA sorok begépelése közben mindenki gyakran vét hibát, hiszen a programozó számára a kódok értelmetlen számok, szemben a BASIC programok utasításszavaival, mint pl. a PRINT vagy az INPUT.
Ebben a könyvben több helyen találunk ezzel a módszerrel készített betöltőprogramot, ilyen pl. az Assembler program betöltője is.

```
0 REM **** P44. ****
1 :
2 :
100 OPEN 1,4 : Z=100
110 INPUT "KEZDOCIM ";A
120 INPUT "VEGCIM   ";E
130 CMD 1 : PRINT Z"FOR I="A" TO "E
140 I=A : Z=Z+10:PRINT Z"READ X:POKE I,X:S=S+X:NEXT"
150 Z=Z+10 : N=0 : PRINT Z"DATA ";
160 X=PEEK(I) : S=S+X : PRINT RIGHT$("   "+STR$(X),3); : N=N+1
170 IF I=E THEN PRINT : GOTO 200
180 I=I+1 : IF N=12 THEN PRINT : GOTO 150
190 PRINT ","; : GOTO 160
200 PRINT Z+10"IF S <>"S" THEN PRINT"CHR$(34)"HIBA A DATASORBAN !"CHR$(34)":END"
210 PRINT Z+20"PRINT"CHR$(34)"OK !!"CHR$(34)
220 PRINT#1, : CLOSE 1
```

11. A COMMODORE 64-ES 6510-ES DISASSEMBLERE

Ebben a fejezetben bemutatunk egy ún. visszafordító vagy disassembler programot. A disassembler feladata az, hogy a tárban elhelyezett gépi kódú programról assembler listát készítsen. Pl. a $A9, $80 kódokból a disassembler előállítja az LDA #$80 utasítást. Futtatása nagyon egyszerű: RUN paranccsal indíthatjuk, majd meg kell adnunk a visszafordítandó gépi kódú program tárbeli kezdő- és végcímét. A program az assembler listát a képernyőre írja, de könnyen átalakíthatjuk – OPEN és CMD utasítások beiktatásával –, ha a listát a nyomtatóra szeretnénk megkapni.

Nézzük meg röviden, hogyan működik a disassembler program. Betölt egy byte-ot a tárból, és azt az assembler utasítás kódjaként értelmezi. A kód alapján egy táblázatban megkeresi a hozzá tartozó assembler utasítást és a címzésmódot. Az utasítás hossza alapján megállapítja, hogy milyen formában kell az operandust kiírnia. Mindezeket a feladatokat egy szubrutin végzi el, amely végezetül a felhasznált változókat is kiírja az assembler listára.

A disassembler programmal nemcsak a saját magunk által elkészített gépi kódú programokról, hanem az interpreter és az operációs rendszer rutinjairól is készíthetünk assembler listát. Ezekből a rutinokból nagyon sok hasznos programozástechnikai ötletet meríthetünk. Még jobb segítséget nyújt az operációs rendszer tanulmányozásához „A Commodore 64-es belső felépítése" c. könyv, amelyben az operációs rendszer teljes, részletesen dokumentált listáját is közreadjuk.

A 6510-es disassembler program listája:

```
0 REM **** P45. ****
1 :
2 :
100 REM 6510 - DISASSEMBLER
110 DIM MN$(255),AD(255),H$(15)
120 FF=2[illegible]:HI=256:UL=2↑16:SC=2↑15-1
130 PRINT "[illegible]6510 - DISASSEMBLER"
140 FOR I=0 TO 15:READ H$(I):NEXT
150 FOR I=0 TO 255:READ MN$(I),AD(I):NEXT
160 PRINT "[illegible]KEZDOCIM    :-    $****[illegible]";:INPUT A$
170 GOSUB 540:S=A
180 PRINT "[illegible]VEGCIM      :-    $****[illegible]";:INPUT A$:PRINT
190 GOSUB 540:E=A
200 FOR P=S TO E
210 A=P:GOSUB 450:REM CIMEK
220 PRINT "  ";:A=PEEK(P):GOSUB 480:PRINT "  ";:I=PEEK(P):OP=AD(I)
230 ON OP GOSUB 510,520,520,510,530,520,520,530,530,520,520,520,530
240 PRINT "  ";MN$(I)"  ";
250 ON OP GOSUB270,280,290,300,310,320,330,340,350,360,370,380,400
260 NEXT P:GOTO 160
270 PRINT:RETURN
280 PRINT "#";:GOSUB 490:P=P+1:PRINT:RETURN
290 GOSUB 490:P=P+1:PRINT:RETURN
300 PRINT " A":RETURN
310 GOSUB 420:P=P+2:PRINT:RETURN
320 GOSUB 490:P=P+1:PRINT ",X":RETURN
330 GOSUB 490:P=P+1:PRINT ",Y":RETURN
340 GOSUB 420:P=P+2:PRINT ",X":RETURN
350 GOSUB 420:P=P+2:PRINT ",Y":RETURN
360 PRINT "(";:GOSUB 490:P=P+1:PRINT "),Y":RETURN
370 PRINT "(";:GOSUB 490:P=P+1:PRINT ",X)":RETURN
380 T=PEEK(P+1):Q=T+HI*(T>127)+2+P
390 A=INT(Q/HI)*HI+((Q+(Q>SC)*UL) AND FF):PRINT"$";:GOSUB 450:P=P+1:PRINT:RETURN
400 PRINT "(";:GOSUB 420
410 PRINT ")":P=P+2:RETURN
```

```
420 PRINT "$";
430 A=PEEK(P+1)+HI*PEEK(P+2)
440 REM HEX HEXACIM A
450 HB=INT(A/HI):A=A-HI*HB
460 PRINT H$(HB/16)H$(HB AND 15);
470 REM HEXADECIMALIS BYTE A
480 PRINT H$(A/16)H$(A AND 15);:RETURN
490 PRINT "$";
500 A=PEEK(P+1):GOTO 480
510 PRINT "       ";:RETURN
520 GOSUB 500:PRINT "    ";:RETURN
530 GOSUB 500:PRINT " ";:A=PEEK(P+2):GOTO 480
540 IF ASC(A$)=42 THEN END
550 A=0:FOR I=1 TO 4:V=ASC(RIGHT$(A$,I))-48:V=V+(V>9)*7:A=A+V*(16↑(I-1)):NEXT:RE
TURN
1000 DATA0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
1010 DATA"BRK", 1,"ORA",11,"???", 1
1020 DATA"???", 1,"???" 1,"ORA", 3
1030 DATA"ASL", 3,"???", 1,"PHP", 1
1040 DATA"ORA", 2,"ASL", 4,"???", 1
1050 DATA"???", 1,"ORA", 5,"ASL", 5
1060 DATA"???", 1,"BPL",12,"ORA",10
1070 DATA"???", 1,"???", 1,"???", 1
1080 DATA"ORA", 6,"ASL", 6,"???", 1
1090 DATA"CLC", 1,"ORA", 9,"???", 1
1100 DATA"???", 1,"???", 1,"ORA", 8
1110 DATA"ASL", 8,"???", 1,"JSR", 5
1120 DATA"AND",11,"???", 1,"???", 1
1130 DATA"BIT", 3,"AND", 3,"ROL", 3
1140 DATA"???", 1,"PLP", 1,"AND", 2
1150 DATA"ROL", 4,"???", 1,"BIT", 5
1160 DATA"AND", 5,"ROL", 5,"???", 1
1170 DATA"BMI",12,"AND",10,"???", 1
1180 DATA"???", 1,"???", 1,"AND", 6
1190 DATA"ROL", 6,"???", 1,"SEC", 1
1200 DATA"AND", 9,"???", 1,"???", 1
1210 DATA"???", 1,"AND", 8,"ROL", 8
1220 DATA"???", 1,"RTI", 1,"ROR",11
1230 DATA"???", 1,"???", 1,"???", 1
1240 DATA"EOR", 3,"LSR", 3,"???", 1
1250 DATA"PHA", 1,"EOR", 2,"LSR", 4
1260 DATA"???", 1,"JMP", 5,"EOR", 5
1270 DATA"LSR", 5,"???", 1,"BVC",12
1280 DATA"EOR",10,"???", 1,"???", 1
1290 DATA"???", 1,"EOR", 6,"LSR", 6
1300 DATA"???", 1,"CLI", 1,"EOR", 9
1310 DATA"???", 1,"???", 1,"???", 1
1320 DATA"EOR", 8,"LSR", 8,"???", 1
1330 DATA"RTS", 1,"ADC",11,"???", 1
1340 DATA"???", 1,"???", 1,"ADC", 3
1350 DATA"ROR", 3,"???", 1,"PLA", 1
1360 DATA"ADC", 2,"ROR", 4,"???", 1
1370 DATA"JMP",13,"ADC", 5,"ROR", 5
1380 DATA"???", 1,"BVS",12,"ADC",10
1390 DATA"???", 1,"???", 1,"???", 1
1400 DATA"ADC", 6,"ROR", 6,"???", 1
1410 DATA"SEI", 1,"ADC", 9,"???", 1
1420 DATA"???", 1,"???", 1,"ADC", 8
1430 DATA"ROR", 8,"???", 1,"???", 1
1440 DATA"STA",11,"???", 1,"???", 1
1450 DATA"STY", 3,"STA", 3,"STX", 3
1460 DATA"???", 1,"DEY", 1,"???", 1
1470 DATA"TXA", 1,"???", 1,"STY", 5
1480 DATA"STA", 5,"STX", 5,"???", 1
1490 DATA"BCC",12,"STA",10,"???", 1
1500 DATA"???", 1,"STY", 6,"STA", 6
1510 DATA"STX", 7,"???", 1,"TYA", 1
```

```
1520 DATA"STA", 9,"TXS", 1,"???", 1
1530 DATA"???", 1,"STA", 8,"???", 1
1540 DATA"???", 1,"LDY", 2,"LDA",11
1550 DATA"LDX", 2,"???", 1,"LDY", 3
1560 DATA"LDA", 3,"LDX", 3,"???", 1
1570 DATA"TAY", 1,"LDA", 2,"TAX", 1
1580 DATA"???", 1,"LDY", 5,"LDA", 5
1590 DATA"LDX", 5,"???", 1,"BCS",12
1600 DATA"LDA",10,"???", 1,"???", 1
1610 DATA"LDY", 6,"LDA", 6,"LDX", 7
1620 DATA"???", 1,"CLV", 1,"LDA", 9
1630 DATA"YSX", 1,"???", 1,"LDY", 8
1640 DATA"LDA", 8,"LDX", 9,"???", 1
1650 DATA"CPY", 2,"CMP",11,"???", 1
1660 DATA"???", 1,"CPY", 3,"CMP", 3
1670 DATA"DEC", 3,"???", 1,"INY", 1
1680 DATA"CMP", 2,"DEX", 1,"???", 1
1690 DATA"CPY", 5,"CMP", 5,"DEC", 5
1700 DATA"???", 1,"BNE",12,"CMP",10
1710 DATA"???", 1,"???", 1,"???", 1
1720 DATA"CMP", 6,"DEC", 6,"???", 1
1730 DATA"CLD", 1,"CMP", 9,"???", 1
1740 DATA"???", 1,"???", 1,"CMP", 8
1750 DATA"DEC", 8,"???", 1,"CPX", 2
1760 DATA"SBC",11,"???", 1,"???", 1
1770 DATA"CPX", 3,"SBC", 3,"INC", 3
1780 DATA"???", 1,"INX", 1,"SBC", 2
1790 DATA"NOP", 1,"???", 1,"CPX", 5
1800 DATA"SBC", 5,"INC", 5,"???", 1
1810 DATA"BEQ",12,"SBC",10,"???", 1
1820 DATA"???", 1,"???", 1,"SBC", 6
1830 DATA"INC", 6,"???", 1,"SED", 1
1840 DATA"SBC", 9,"???", 1,"???", 1
1850 DATA"???", 1,"SBC", 8,"INC", 8
1860 DATA"???", 1
```

Programleírás:

100–150 A kezdőértékek beállítása (inicializálás), a tömbök felépítése

160–190 A kezdő- és végcím beolvasása

200–260 A disassembláló FOR–NEXT ciklus a kezdőcímtől a végcímig. Operandussal nem rendelkező utasítás esetén a programszámláló értéke automatikusan eggyel nő. A 220-as sorban beolvassuk az utasításkódot és kiírjuk az aktuális címet. A 230-as sorban a címzésmódnak megfelelően kiírjuk az operandusnak megfelelő byte-ot. A 240-es sorban kiírjuk az utasításszót, a 250-es sorban pedig az operandust. A 260-as sorban lezárjuk a ciklust és visszaugrunk a beolvasásra.

280–410 Az operandus kiírása a címzésmódnak megfelelően.

420–530 A byte-okat és címeket hexadecimális alakban kiíró szubrutin.

540–550 A hexadecimális számok átalakítása decimálissá.

1000–1860 Az utasításszavak és címzésmódok táblázata.

A program változói:

MN$(255)	Az utasításszavak tömbje
AD(255)	A címzésmódok tömbje
H$(15)	Hexadecimális számjegyek
FF	Konstans 255
HI	Konstans 255
UL	Konstans 65536
SC	Konstans 32767
A$	A hexadecimális számokat tartalmazó füzér
S	Kezdőcím
E	Végcím
P	Utasításszámláló
OP	Címzésmód

12. A GÉPI KÓDÚ PROGRAMOK KÉSZÍTÉSE FEJLETT PROGRAMOZÁSTECHNIKÁVAL

Az előző fejezetekben ismertettünk egy BASIC nyelven megírt Assembler programot, amely a tárban kb. 11 kbyte-ot foglal el. Ez az Assembler program a forrásprogramot két menetben fordítja le, és megengedi a szimbolikus címek és változók alkalmazását. A könyvben bemutatott viszonylag kis méretű programok lefordítására ez az Assembler tökéletesen megfelelt. Ha azonban nagyobb terjedelmű, bonyolultabb programokat készítenénk, hamarosan beleütköznénk az ismertetett fordítóprogram korlátaiba. Hosszabb forrásprogram esetében különösképpen a fordítási idő szembetűnő megnövekedése okozna problémát. Több mint száz sorból álló forrásprogram lefordítása alaposan próbára tenné a türelmünket, különösen akkor, ha a forrásprogramban az Assembler még hibát is találna. Ekkor ugyanis a hiba kijavítása után a fordítást meg kellene ismételni, azaz a forrásprogramot újra be kellene tölteni, az egész eljárást elölről kezdeni, miközben semmi biztosítékunk nincs arra nézve, hogy a program most már hibátlan.
Összefoglalva a tapasztalatokat azt mondhatjuk, hogy a könyvbeli Assembler program oktatási célokra kitűnő, de összetettebb feladatok megoldására nem alkalmas. Nem lesz tehát haszontalan, ha ismertetünk egy olyan assembler programot a C 64-esen, amely gépi kódban készült, és annak ellenére, hogy a tárban mindössze 8 kbyte-ot foglal el, sokkal hatékonyabb, mint a BASIC nyelvű assembler. A forrásprogramot az eddigiekhez hasonlóan kell elkészíteni, de nem kell fordítás előtt a lemezen tárolni. Ez a program, melyet egyszerű SYS utasítással indíthatunk, néhány másodperc alatt lefordítja azt a gépi kódú programot, amelynek lefordítása a könyvben ismertetett Assemblerrel perceket vett igénybe. Foglaljuk össze a gépi kódú assembler alapvető sajátosságait.

A PROFI–ASS 64 2.0 egy kétmenetes 6510-es és 6502-es MACRO-Assembler a C 64-esen. Maga a program gépi kódban készült, 8 kbyte-os, és lemezről kell betölteni. A forrásprogramot a megszokott módon, a BASIC editor segítségével kötetlen formában begépelhetjük a billentyűzeten. Fordítás után kapunk egy teljes assembler listát, egy betölthető szimbólumtáblázatot, tároljuk a gépi kódú modult (opcode-ot), dolgozhatunk újra definiálható szimbólumokkal, és egy sereg különböző assembler utasítással. Az utasítások formátuma emlékeztet az ún. MOS-standard formátumra. Az assembler forrásprogramot, a BASIC programokhoz hasonlóan, soronként be kell gépelni. Az assembler sorokat a BASIC-hez hasonlóan kijavíthatjuk, a felesleges sort törölhetjük, vagy új sort illeszthetünk a programba. A programírás nem igényel saját szerkesztőprogramot, így a forrásprogram nagyobb területet foglalhat el, összesen 34 kbyte-ot. Egy sorba, kettősponttal elválasztva több assembler utasítást írhatunk, éppúgy mint BASIC-ben.
Az assembler forrásprogram szimbólumokból, utasításszavakból (mnemonikokból), operandusokból és megjegyzésekből épül fel. Ezeken kívül használhatunk még néhány ún. pszeudoműveletet, amelyek nem gépi kódú utasítások, hanem a PROFI–ASS 64 2.0 speciális feladatokat elvégző utasításai.
Az utasításokat elláthatjuk címkékkel. Ha a sor tartalmaz címkét, azt néhány üres karakterrel kell elválasztani az utasítástól. A címkét mindig betűvel kell kezdeni, a többi karaktere lehet bármilyen betű vagy szám. Két címke azonosnak számít, ha az első nyolc karakterük azonos. Különleges karakterek (kettőspont, felkiáltójel stb.) nem szerepelhetnek a címkékben.
Az utasításszavakat vagy a címke után, vagy ha az adott sorban nincs címke, a sor elején kell elhelyeznünk. Minden utasításszó három betűből áll. Ezeket nem szabad címkeként használni. A pszeudoutasításokat ponttal, vagy egy különleges karakterrel kell elválasztani

az operandusuktól, kivéve az "=" és a "*=" utasításokat. Azokat a pszeudoutasításokat, amelyek ponttal kezdődnek, a fordítóprogram az első három karakterük alapján különbözteti meg egymástól, de az assembler listára kiírja azok teljes nevét.
Az assembler sort egy pontosvesszővel zárhatjuk le, a pontosvesszőt követő szöveg megjegyzésnek számít, amit a program a fordítás közben nem vesz figyelembe. A szövegben előforduló kettőspontot a fordító az utasítás végét jelző karakternek tekinti, hacsak a szöveg nincs idézőjelek közé zárva. A pontosvesszővel kezdődő sort a fordító megjegyzésnek tekinti.
Az operandusmező tartalmazza a címzésmódot és az utasításhoz vagy pszeudokódhoz szükséges kifejezést.

A címzésmódok szintaxisa:

# kifejezés		közvetlen címzés
kifejezés		abszolút vagy relatív címzés
kifejezés, X	abszolút, X	X-szel indexelt címzés
kifejezés, Y	abszolút, Y	Y-nal indexelt címzés
(kifejezés, X)		indexelt indirekt címzés
(kifejezés, Y)		indirekt indexelt címzés
(kifejezés)		indirekt címzés

Ha a kifejezés értéke kisebb mint 256, a PROFI–ASS 64 2.0 az abszolút címzés helyett nulláslap címzéssel dolgozik. A könyvben közölt BASIC nyelvű Assembler ezt nem teszi meg automatikusan. Ott, ha nulláslap címzéssel akarunk dolgozni, azt egy csillag karakterrel kell az assembler tudomására hoznunk. A PROFI–ASS 64 is kényszeríthető arra, hogy 256-nál kisebb címek esetén is abszolút címzést használjon. Míg az LDA !5,X lefordított alakja BD 05 00, ami az LDA utasítás abszolút címzésű formája, addig az LDA 5,X utasítást a fordító a nulláslap címzésű B5 05 kóddá alakítja át.

Kifejezések

A PROFI–ASS 64 2.0 program egyik erőssége, hogy összetett kifejezéseket képes értelmezni. A programban van egy rekurzív rutin, amely kiértékeli a tetszőleges mélységig egymásba ágyazott kifejezéseket, és ezzel jóval szélesebb lehetőséget kínál használójának, mint a MOS Standard vagy más assembler programok.
Az assembler forrásprogramban mindenütt használhatunk a konstansok helyett kifejezéseket, ahogyan az a fenti felsorolásból is látható.
A forrásprogramokat a kifejezések használata áttekinthetővé és könnyen javíthatóvá teszi.
A kifejezések szintaxisa nagyon egyszerű, a MOS Standardban megengedett alakok itt is megengedettek. A kifejezéseket úgy kell megadni, mintha zsebszámológéppel dolgoznánk. A feldolgozás mindig balról jobbra halad, hacsak kerek zárójellel nem szabtuk meg a műveletek sorrendjét.

A rendelkezésre álló műveletek:

+	összeadás
–	kivonás (a bal oldali operandusból kivonjuk a jobb oldalit)
*	szorzás
/	osztás (a bal oldali operandust osztjuk a jobb oldalival)

!	logikai OR művelet
&	logikai AND művelet
↑	logikai EOR (kizáró vagy) művelet
>	a jobb oldali argumentum által megadott számú biteltolás jobbra
<	a jobb oldali argumentum által megadott számú biteltolás balra

A program minden műveletet 16 biten végez el, ha mégis túlcsordulás adódna (pl. ha 32767-nél nagyobb számértékkel szorzunk, vagy több mint 15 bittel balra eltolunk egy számot), megjelenik az ILLEGAL QUANTITY ERROR hibaüzenet.
Összeadásnál és kivonásnál a fordító a 65535-nél nagyobb számértéket negatív szám kettes komplemenseként értelmezi.
Az operandusokat többféle alakban megadhatjuk. A szintaxist a következő felsorolás tartalmazza:

Operandus-típusok

Típus	Példa
Hexadecimális	$1C3
Decimális	127
Bináris	%110011
PC	*
ASCII karakter	"A"
Címke	SYMB
Kifejezés	("Z"+6)

Az operandusokat a megengedett műveletek bármelyike összekapcsolhatja. A feldolgozás sorrendjét zárójelezéssel szabályozhatjuk. Minden operandus előtt – kivéve a zárójeles kifejezést – állhat egy negatív előjel, amely az assembler számára azt jelenti, hogy a számérték kettes komplemensét kell venni. A kifejezések előtt állhatnak ún. módosító karakterek. Ezek közül a felkiáltójel jelentésére már utaltunk. A nagyobb (>) és kisebb (<) relációjelekkel leválaszthatjuk a felső, ill. az alsó byte-ot. Erre a közvetlen címzésnél vagy a .BYTE pszeudoutasításnál lehet szükség.
A felső byte operátor (>) hatása ua., mintha a

Kifejezés > 8

műveletet, az alsó byte operátor (&) hatása pedig ua. mintha a

Kifejezés & $FF

műveletet végeztük volna el.

Pszeudoműveletek

.BYTE

A .BYTE művelettel egy 1 byte-os értéket elhelyezhetünk a programszámlálóba. Ez a művelet megfelel a könyvbeli Assembler program .BY műveletének. A művelet után, attól vesszővel elválasztva, operandusokként érvényes PROFI–ASS 64 2.0 kifejezést írhatunk. A kifejezés hosszát csak a sor hossza és a puffer mérete korlátozza. A kifejezés értéke csak 1 byte-os számérték lehet, egyébként az ILLEGAL QUANTITY ERROR hibaüzenetet kapjuk. Két byte-os értékekkel is dolgozhatunk, ha > vagy < módosító karakterekkel leválasztjuk a felső, ill. az alsó byte-ot.
Az 1 byte-os érték a 0-tól 255-ig, ill. $FF80-tól $FFFF-ig terjedő számtartományba esik. A .BYTE-1 utasítás is megengedett, a számtartomány felső felét ugyanis a fordító negatív számként értelmezi. A .BYTE utasítást ugrási címtáblázatok vagy mutatók meghatározására használhatjuk. Így olyan utasításokat is „becsempészhetünk" a programba, amelyeket közvetlenül a fordító nem tudna feldolgozni; ilyen pl. a BIT utasítás:

```
      .BYTE $2C ;  abszolút BIT utasítás
CÍMKE LDA#-1 → ;  rejtett LDA utasítás
```

.WORD

A .WORD utasítással a két byte-os címeket a szokásos alakban, alsó, felső byte-ra bontva helyezhetjük el a forrásprogramban. Az eredmény azonos a .WORD CÍM utasítás eredményével.

.FILE

A .FILE utasítással több forrásprogramot összeláncolhatunk. Az utasítás szintaxisa:

.FILE egységszám, "file-név"

Az egységszám az aktuális lemezegység vagy kazettás egység száma, a file-név pedig az egységről betöltendő file neve. Hosszabb assembler programokat egységenként is elkészíthetünk, és az egyes egységeket a .FILE utasítással összefűzhetjük egy programmá.

.IF

A .IF a feltételes elágazás szervezésére használható. Az utasításban argumentumként megadhatunk egy kifejezést, amelynek értékét a fordító az első és a második menetben kiszámítja. Ha a kifejezés értéke nulla, a program a .IF utasítás sorában álló kódot assemblálja. Ebben a sorban általában .GOTO utasítás áll, azaz egy elágazás a program valamely sorára. A sorban elhelyezett kódot kettősponttal kell lezárni. A .IF és a .GOTO utasításokkal, ill. a szimbólumok újradefiniálásával a forrásprogramba ciklusokat is beépíthetünk. Bár a .IF utasítás csak az argumentum nulla értékét vizsgálja, egy kis trükkel a vizsgálatot tetszőleges számértékre kiterjeszthetjük. Ha egy számot 15 bittel eltolunk jobbra, az eredmény 1 vagy 0 aszerint, hogy a szám eredetileg negatív vagy pozitív volt. Ha tetszőleges kifejezések értékét akarjuk összehasonlítani, ki kell vonnunk őket egymásból, és a fenti módszerrel meg kell vizsgálni, hogy az eredmény pozitív vagy negatív.

.GOTO

A .GOTO utasítás egy feltétlen ugrás az argumentumként megadott sorra. Az argumentum itt is lehet kifejezés. A .IF utasítással együtt a .GOTO a ciklusszervezés eszköze.

.GTB

Az utasításjel a Go To BASIC szöveg rövidítése. Argumentuma nincs, egyszerűen visszaadja a vezérlést a BASIC programnak. Az assembler programba a BASIC-ből egy speciális ugrási címmel térhetünk vissza.

.ASC "SZOVEG"

Ezzel az utasítással szövegeket illeszthetünk a forrásprogramba. Egyik felhasználása lehet például a file nevének elhelyezése az OPEN utasításban.

.SYS

Ezzel az utasítással meghívhatunk egy saját gépi kódú programot fordítás közben. Az utasítás argumentuma tetszőleges kifejezés lehet, amelyet a fordító ugrási címként értelmez. A .SYS utasítás működése hasonló a BASIC nyelvbeli SYS utasítás működéséhez. Azok a programozók, akik jól ismerik a PROFI–ASS 64 2.0 programot, a .SYS utasítással saját pszeudoutasítás készletet alakíthatnak ki.

.SST egységszám, másodlagos cím, „file-név"

A forrásprogramban használt szimbólumok táblázatát tárolhatjuk az IEC-Buszra csatlakoztatott külső egységen, pl. a lemezegységen. A .SST utasítást a fordító az első menetben értelmezi, és hatására az eddig generált szimbólumot tárolja a lemezen.
Az utasítás első operandusa az egységszám, ami általában 8. A második operandus, a másodlagos cím 2 és 14 közé kell hogy essen. A file nevét mindig meg kell adni, és a név mögé a ",S,W" karaktereket kell elhelyezni.
A .SST utasításra akkor lehet szükségünk, ha a szimbólumokról és a címkékről egy rendezett listát szeretnénk készíteni.

.LST egységszám, másodlagos cím, "file-név"

A .LST utasítás a .SST utasítás ellentettje. A lemezen tárolt szimbólumtáblázatot betölti a tárba.

.END

A .END utasítás a forrásprogram végét jelzi. Hatása a második menet végén azonos a .GTB utasítás hatásával, azaz a vezérlés visszakerül a BASIC programra, ahonnan azonnal meghívhatjuk az éppen most lefordított forrásprogramot SYS utasítással.

.OPT

Ezzel az utasítással kérhetünk a fordítótól assembler, ill. gépi kódú (object code) listát. Az utasítás szintaxisa:

.OPT opció, opció, opció,...

A következő lehetőségek közül választhatunk:

P – Print. A P opcióval dönthetünk arról, hogy a listát melyik egységre kérjük. Ha csak egy P-t írunk az utasítás mögé, a listát a képernyőre kapjuk. Ha a nyomtatóra szeretnénk dolgozni, előzetesen a BASIC OPEN utasítással meg kell nyitnunk a nyomtatót, pl. OPEN 1,4, majd .OPT P1 utasítással kérhetjük a programlistát.

O – Object. Ezzel az opcióval eldönthetjük, hogy az előállított gépi kódú programot hová helyezze a fordítóprogram. Ha .OPT 00 utasítást adunk, a gépi kódú program egyenesen a tárba kerül (mint ahogyan a könyvbeli Assembler program esetében). A .OPT 01 utasítás eredményeként a gépi kód az 1-es logikai számmal ellátott file-ban tárolódik. Ha fordítás előtt az OPEN 1,8,1,"0:PROGRAM" utasítással megnyitottunk egy file-t, a gépi kód a lemezes file-ba kerül, ahonnan LOAD utasítással bármikor betölthető.

A következő mintaprogram a nulláslap tartalmát a megadott sortól kezdve kiírja a képernyőre.

```
0 REM **** P46. ****
1 :
2 :
10 SYS 32768 : REM A PROFI ASS 64 HIVASA
20 .OPT P,00 ; LISTAZAS A KEPERNYORE, OBJEKT A TARBAN
30 *= $C000 ; PRG. KEZDOCIM
40 LINE = 10 ; KEPERNYORE
50 VIDEO = $400 ; A KEPERNYOTAR CIME
60 COLOR = $D800 ; A SZIN RAM CIME
70 SZIN = 1 ; AZ IRAS FEHER
80 LDX #0
90 CIKLUS LDA 0,X ; A 0-AS LAPROL 1 BYTE BETOLTESE
100 STA VIDEO+(40*LINE),X ; A JEL KIIRASA
110 LDA #SZIN
120 STA COLOR+(40*LINE),X ; SZINBEALLITAS
130 INX
140 BNE CIKLUS
150 RTS
160 .EN
```

A következő példában a gépi kódot közvetlenül lemezen tároljuk, és a listát a nyomtatóra kérjük. A forrásprogram több egységből áll.

```
0 REM **** P47. ****
1 :
2 :
10 OPEN 1,8,1,"0:OBJEKTKOD"
20 OPEN 2,4:REM NYOMTATO"
30 SYS 32768
40 .OPT 01,P2
50 ; ASSEMBLERUTASITASOK
```

```
...
1000 .FILE 8, "2.PROGRAM"

2. program
   10; további utasítások
...
1000 .FILE 8, "3.PROGRAM"

3. program
   10; további utasítások
...
1000 .END 8, "1.PROGRAM"
```

Az 1-es program tartalmazza a SYS 32768 utasítást. A PROFI–ASS 64 2.0 az esetleges hibaüzenetet a hibás sor előtt világos betűkkel írja ki.

A PROFI–ASS 64 assembler a Commodore 64-esre készített PROFIMAT szoftvertermék egyik programja. A PROFIMAT az assembler programon kívül tartalmaz még egy kényelmes monitorprogramot, a PROFI–MON 64-et, amelynek működését az alábbiakban ismertetjük.
Arról, hogy mi a feladata egy monitorprogramnak, már korábban, a gépi kódú programok beviteléről szóló fejezetben ejtettünk néhány szót. A monitorprogram elsősorban a tár és a regiszterek tartalmának megjelenítésére szolgál. Ugyanakkor felhasználásával lemezen tárolhatjuk, ill. a tárba betölthetjük a gépi kódú programokat. A gépi kódú programok futtathatóak is a monitorprogramból, és ha a gépi kódú programot BRK utasítással zárjuk, végrehajtás után a vezérlés ismét visszakerül a monitorprogramhoz, és a képernyőn megjelenik a regiszterek tartalma.
A fentieken kívül a PROFI–MON 64 rendelkezésünkre bocsát egy sereg kényelmes utasítást, amelyek jelentős mértékben megkönnyítik a gépi kódú programozást, és a gépi kódú programok tesztelését.
A következőkben bemutatjuk a PROFI–MON 64 utasításkészletét, és az utasítások beírásának módját. A programot a LOAD "PROFI MON 64", 8,1 utasítással kell a tárba betölteni, és betöltés után a SYS 49152 utasítással kell elindítani. A program bejelentkezése:

```
C*
     PC   IRQ       SR  AC  XR  YR  SP  NV-BDIZC
> ;  0003 EA31      32  34  02  A2  F8  00110010
```

A C betű a Call (hívás) angol szó kezdőbetűje. A következő sorokban megjelenik a képernyőn a regiszterek tartalma, ahogyan azt már az egylépéses szimulátornál is láttuk. A jelölések ugyanazok, mint korábban. Az IRQ (interrupt request) a megszakítási vektort jelöli, amelynek tartalma az a cím, ahová a program elágazik, ha a megszakítási vezetéken impulzust észlel.
A regiszterek tartalmát megváltoztathatjuk úgy, hogy a kurzort a megfelelő helyre mozgatjuk, az ott található értéket felülírjuk, majd megnyomjuk a RETURN billentyűt. A kapcsolókat az állapotregiszteren keresztül módosíthatjuk. Az állapotregiszter tartalmát megváltoztatva, a kapcsolók értéke is automatikusan módosul.
A regiszterek tartalmát a

```
>R
```

utasítással és a RETURN billentyű megnyomásával írathatjuk a képernyőre.
A következő utasítással az egyes tárcímek tartalmát írathatjuk ki és változtathatjuk meg. Az utasítás jele az M betű, amelyet a látni kívánt tárterület kezdő- és végcíme követ, hexadecimális alakban. Például:

```
> M A0A0 A0AF
> :  A0A0  C4  46  4F  D2  4E  45  58  D4  DFORNEXT
> :  A0A8  44  41  54  C1  49  4E  50  55  DATAINPU
```

A > jelet a PROFI–MON 64 írja ki a képernyőre, ezzel jelezve, hogy a program a gépkezelő utasítására vár.
A képernyő tartalmát a következőképpen értelmezhetjük. A kettőspont után a sorban kijelzett nyolc byte közül az első címe áll. A mi példánkban a $A0A0 tárcím tartalma $C4. A $A0A1 cím tartalma $46 stb. A sor végén láthatóak a kódoknak megfelelő ASCII karakterek. Ezeket a program reverse módban írja a képernyőre. Ha az adott tárcímen egy nem nyomtatható karakter kódja található, a program a karakter helyére egy pontot ír.
A tárcímek tartalmát a szokott módon változtathatjuk meg. A kurzort az adott helyre mozgatjuk, az aktuális értéket felülírjuk, majd megnyomjuk a RETURN billentyűt.
A gépi kódú program indítására szolgál a G parancs. Ha a program pl. a $CF20-as tárcímen kezdődik, a

```
>G CF20
```

paranccsal indíthatjuk. Végrehajtás előtt R paranccsal kiírathatjuk, és megfelelőképpen beállíthatjuk a regiszterek tartalmát. Ha a gépi kódú program BRK utasítással végződött, végrehajtás után a vezérlés ismét a monitorprogramra kerül, és a képernyőn automatikusan megjelenik a regiszterek aktuális tartalma, például:

```
B*
   PC    IRQ   SR  AC  XR  YR  SP  NV–BDIZG
>; CF39  EA31  B3  8F  73  B0  F6  10110011
```

A kiírásban a B betű arra utal, hogy a gépi kódú program utolsó utasítása BRK volt. A programszámláló a BRK utasítást követő utasítás címét tartalmazza. Ebből következtethetünk arra, hogy melyik programrészben volt megszakítás, abban az esetben, ha több BRK utasítást építettünk be a programba. Célszerű a nagyobb terjedelmű, bonyolultabb programok tesztelését azzal megkönnyíteni, hogy minden kritikusabb programegységben elhelyezünk egy-egy BRK utasítást.
Megszakításkor a program működése szempontjából fontosabb regiszterek tartalmát ellenőrizhetjük. Ha egy programrész már hibátlanul működik, az ideiglenesen beépített BRK utasítást helyettesíthetjük a tényleges ugrási címmel, és a BRK-t áthelyezhetjük a következő kritikus programegység végére.
Így lépésenként könnyűszerrel elkészíthetjük a hibátlanul működő teljes programot.
A gépi kódú programok betöltésére és tárolására szolgál az L, ill. az S parancs.
A betöltő parancs szintaxisa:

```
>L "NEV" ,XX
```

Az idézőjelek közé a program nevét, a vessző után pedig az egységszámot (hexadecimális alakban) kell beírni.
Ha például a lemezről a GRAFIKA nevű programot akarjuk betölteni, a megfelelő utasítás.

```
>L "GRAFIKA" ,08
```

Ha a tároló kazettás egység, akkor az egységszám 01. Betöltéskor a parancs további paramétereként megadhatjuk a betöltési tárcímet. Ha például a GRAFIKA programot a tárban a $1000-es címtől kezdve akarjuk elhelyezni, a megfelelő parancs:

```
>L "GRAFIKA" ,08,1000
```

Az S (SAVE) parancs hasonlóképpen működik. Természetesen mentésnél meg kell adni a gépi kódú program tárbeli kezdő- és végcímét. A végcímnek mindig eggyel nagyobbnak kell lennie, mint a tényleges program végcíme. Az alábbi utasítás:

```
>S "PROGRAM" ,08,7000,8000
```

a $7000-tól $7FFF-ig terjedő tárterület tartalmát menti a lemezre. Kazettás tárolásnál az egységszám itt is 01.

A PROFI–MON 64 további szolgáltatása a beépített disassembler program. A D paranccsal egy tetszőleges tártartományon található gépi kódú program assembler listáját kiírathatjuk a képernyőre. Például a

```
>D B824 B82C
```

utasítás hatására a következő lista jelenik meg a képernyőn:

```
>,  B824  20  EB  B7  JSR  $B7EB
>,  B827  8A          TXA
>,  B828  A0  00      LDY  #$00
>,  B82A  91  14      STA  ($14),Y
>,  B82C  60          RTS
```

A következő paranccsal összehasonlíthatjuk két tárterület tartalmát. Ha két tárcím tartalma eltér egymástól, a monitorprogram ezek közül az egyiket kiírja a képernyőre. Például a

```
>C 8000 8100 9000
```

utasítás hatására a monitorprogram összehasonlítja a $8000-tól $8100-ig terjedő tárcímek tartalmát a $9000-tól $9100-ig terjedő tárcímek tartalmával. A futás eredményeként a képernyőn megjelenik a

```
9056
```

ami azt jelenti, hogy a $8056-os cím tartalma nem azonos a $9056-os cím tartalmával.

A T paranccsal átmásolhatjuk valamely tárterület tartalmát egy másik területre. Itt is meg kell adnunk a másolandó terület kezdő- és végcímét, illetve a másolat kezdőcímét. A másolandó terület és a regiszterek tartalma másolás közben nem változik meg. A

>T 6000 6FFF 3000

utasítás hatására a monitorprogram átmásolja a $6000-tól $6FFF-ig terjedő tárterület tartalmát a $3000-as címtől $3FFF-ig terjedő területre.

A monitorprogrammal a tár egy meghatározott területén megkereshetjük azt a tárcímet, ahol egy általunk megadott kódsorozat található. A keresést a H paranccsal végezzük. Például a

>H E000 EFFF 20 D2 FF

paranccsal a kijelölt területen a $20, $D2, $FF byte-sorozatot keressük, ami a JSR $FFD2 utasítás kódja. A monitorprogram kiírja válaszul azt a tárcímet (vagy tárcímeket), ahol ilyen kódsorozatot talált.
A H paranccsal szöveget is kereshetünk a tárban. A keresett szöveget idézőjelek között kell megadni.

>H A000 AFFF "READY"

A READY szó a tár $A378-as címén található, a fenti parancs hatására a monitorprogram ezt a címet fogja a képernyőre kiírni.

Az utolsó parancs a tárterület feltöltésére szolgál. Ha a tár egy tartományát konstansokkal akarjuk feltölteni, az F parancsot kell használnunk. Például a

>F 8000 8FFF 00

parancs hatására a monitorprogram a kijelölt tárterület minden byte-jába 00-t ír.

A monitorprogramból a >X paranccsal léphetünk ki. A

>X

parancs hatására a program visszaadja a vezérlést a BASIC interpreternek, ami a READY üzenettel jelentkezik a képernyőn.

Ha később ismét használni akarjuk a monitorprogramot, a

SYS 49152

parancsot kell begépelnünk.

13. ÁTSZÁMÍTÁSI ÉS UTASÍTÁSTÁBLÁZATOK

Címzésmódok és utasításkódok

	A	#	ZP	AB	ABX	ABY	ZPX	ZPY	.X)	).Y
ADC	–	69	65	6D	7D	79	75	–	61	71
AND	–	29	25	2D	3D	39	35	–	21	31
ASL	0A	–	06	0E	1E	–	16	–	–	–
BIT	–	–	24	2C	–	–	–	–	–	–
CMP	–	C9	C5	CD	DD	D9	D5	–	C1	D1
CPX	–	E0	E4	EC	–	–	–	–	–	–
CPY	–	C0	C4	CC	–	–	–	–	–	–
DEC	–	–	C6	CE	DE	–	D6	–	–	–
EOR	–	49	45	4D	5D	59	55	–	41	51
INC	–	–	E6	EE	FE	–	F6	–	–	–
LDA	–	A9	A5	AD	BD	B9	B5	–	A1	B1
LDX	–	A2	A6	AE	–	BE	–	B6	–	–
LDY	–	A0	A4	AC	BC	–	B4	–	–	–
LSR	4A	–	46	4E	5E	–	56	–	–	–
ORA	–	09	05	0D	1D	19	15	–	01	11
ROL	2A	–	26	2E	3E	–	36	–	–	–
ROR	6A	–	66	6E	7E	–	76	–	–	–
SBC	–	E9	E5	ED	FD	F9	F5	–	E1	F1
STA	–	–	85	8D	9D	99	95	–	81	91
STX	–	–	86	8E	–	–	–	96	–	–
STY	–	–	84	8C	–	–	94	–	–	–

elágazó utasítások	BPL	BMI	BVC	BVS	BCC	BCS	BNE	BEQ
	10	30	50	70	90	B0	D0	F0
eltolási utasítások	TXA	TAX	TYA	TAY	TSX	TXS		
	8A	AA	98	A8	BA	9A		
veremutasítások	PHP	PLP	PHA	PLA				
	08	28	48	68				
ugróutasítások	BRK	JSR	RTI	RTS	JMP	JMP	NOP	
	00	20	40	60	4C	6C	EA	
kapcsolóutasítások	CLC	SEC	CLI	SEI	CLV	CLD	SED	
	18	38	58	78	B8	D8	F8	
számlálóutasítások	DEY	INY	DEX	INX				
	88	C8	CA	E8				

A számok decimális, hexadecimális és bináris alakja

DEC	HEX	BIN	DEC	HEX	BIN
0	00	00000000	43	2B	00101011
1	01	00000001	44	2C	00101100
2	02	00000010	45	2D	00101101
3	03	00000011	46	2E	00101110
4	04	00000100	47	2F	00101111
5	05	00000101	48	30	00110000
6	06	00000110	49	31	00110001
7	07	00000111	50	32	00110010
8	08	00001000	51	33	00110011
9	09	00001001	52	34	00110100
10	0A	00001010	53	35	00110101
11	0B	00001011	54	36	00110110
12	0C	00001100	55	37	00110111
13	0D	00001101	56	38	00111000
14	0E	00001110	57	39	00111001
15	0F	00001111	58	3A	00111010
16	10	00010000	59	3B	00111011
17	11	00010001	60	3C	00111100
18	12	00010010	61	3D	00111101
19	13	00010011	62	3E	00111110
20	14	00010100	63	3F	00111111
21	15	00010101	64	40	01000000
22	16	00010110	65	41	01000001
23	17	00010111	66	42	01000010
24	18	00011000	67	43	01000011
25	19	00011001	68	44	01000100
26	1A	00011010	69	45	01000101
27	1B	00011011	70	46	01000110
28	1C	00011100	71	47	01000111
29	1D	00011101	72	48	01001000
30	1E	00011110	73	49	01001001
31	1F	00011111	74	4A	01001010
32	20	00100000	75	4B	01001011
33	21	00100001	76	4C	01001100
34	22	00100010	77	4D	01001101
35	23	00100011	78	4E	01001110
36	24	00100100	79	4F	01001111
37	25	00100101	80	50	01010000
38	26	00100110	81	51	01010001
39	27	00100111	82	52	01010010
40	28	00101000	83	53	01010011
41	29	00101001	84	54	01010100
42	2A	00101010	85	55	01010101

DEC	HEX	BIN	DEC	HEX	BIN
86	56	01010110	133	85	10000101
87	57	01010111	134	86	10000110
88	58	01011000	135	87	10000111
89	59	01011001	136	88	10001000
90	5A	01011010	137	89	10001001
91	5B	01011011	138	8A	10001010
92	5C	01011100	139	8B	10001011
93	5D	01011101	140	8C	10001100
94	5E	01011110	141	8D	10001101
95	5F	01011111	142	8E	10001110
96	60	01100000	143	8F	10001111
97	61	01100001	144	90	10010000
98	62	01100010	145	91	10010001
99	63	01100011	146	92	10010010
100	64	01100100	147	93	10010011
101	65	01100101	148	94	10010100
102	66	01100110	149	95	10010101
103	67	01100111	150	96	10010110
104	68	01101000	151	97	10010111
105	69	01101001	152	98	10011000
106	6A	01101010	153	99	10011001
107	6B	01101011	154	9A	10011010
108	6C	01101100	155	9B	10011011
109	6D	01101101	156	9C	10011100
110	6E	01101110	157	9D	10011101
111	6F	01101111	158	9E	10011110
112	70	01110000	159	9F	10011111
113	71	01110001	160	A0	10100000
114	72	01110010	161	A1	10100001
115	73	01110011	162	A2	10100010
116	74	01110100	163	A3	10100011
117	75	01110101	164	A4	10100100
118	76	01110110	165	A5	10100101
119	77	01110111	166	A6	10100110
120	78	01111000	167	A7	10100111
121	79	01111001	168	A8	10101000
122	7A	01111010	169	A9	10101001
123	7B	01111011	170	AA	10101010
124	7C	01111100	171	AB	10101011
125	7D	01111101	172	AC	10101100
126	7E	01111110	173	AD	10101101
127	7F	01111111	174	AE	10101110
128	80	10000000	175	AF	10101111
129	81	10000001	176	B0	10110000
130	82	10000010	177	B1	10110001
131	83	10000011	178	B2	10110010
132	84	10000100	179	B3	10110011

DEC	HEX	BIN	DEC	HEX	BIN
180	B4	10110100	218	DA	11011010
181	B5	10110101	219	DB	11011011
182	B6	10110110	220	DC	11011100
183	B7	10110111	221	DD	11011101
184	B8	10111000	222	DE	11011110
185	B9	10111001	223	DF	11011111
186	BA	10111010	224	E0	11100000
187	BB	10111011	225	E1	11100001
188	BC	10111100	226	E2	11100010
189	BD	10111101	227	E3	11100011
190	BE	10111110	228	E4	11100100
191	BF	10111111	229	E5	11100101
192	C0	11000000	230	E6	11100110
193	C1	11000001	231	E7	11100111
194	C2	11000010	232	E8	11101000
195	C3	11000011	233	E9	11101001
196	C4	11000100	234	EA	11101010
197	C5	11000101	235	EB	11101011
198	C6	11000110	236	EC	11101100
199	C7	11000111	237	ED	11101101
200	C8	11001000	238	EE	11101110
201	C9	11001001	239	EF	11101111
202	CA	11001010	240	F0	11110000
203	CB	11001011	241	F1	11110001
204	CC	11001100	242	F2	11110010
205	CD	11001101	243	F3	11110011
206	CE	11001110	244	F4	11110100
207	CF	11001111	245	F5	11110101
208	D0	11010000	246	F6	11110110
209	D1	11010001	247	F7	11110111
210	D2	11010010	248	F8	11111000
211	D3	11010011	249	F9	11111001
212	D4	11010100	250	FA	11111010
213	D5	11010101	251	FB	11111011
214	D6	11010110	252	FC	11111100
215	D7	11010111	253	FD	11111101
216	D8	11011000	254	FE	11111110
217	D9	11011001	255	FF	11111111

A 6510-es utasítások bitmintája és hatásuk a kapcsolókra

*	BITMINTA	N	V	B	D	I	Z	C	oldalszám
ADC	011XXX01	X	X				X	X	23
AND	001XXX01	X					X		26
ASL	000XXX10	X					X	X	37
BCC	10010000								33
BCS	10110000								33
BEQ	11110000								33
BIT	0010X100	M	M				X		28
BMI	00110000								33
BNE	10010000								33
BPL	00010000								33
BRK	00000000			1		1			41
BVC	01010000								33
BVS	01110000								33
CLC	00011000							0	35
CLD	11011000				0				36
CLI	01011000					0			36
CLV	10111000		0						36
CMP	110XXX01	X					X	X	29
CPX	1110XX00	X					X	X	30
CPY	1100XX00	X					X	X	30
DEC	110XX110	X					X		35
DEX	11001010	X					X		35
DEY	10001000	X					X		35
EOR	010XXX01	X					X		28
INC	000XX110	X					X		34
INX	11101000	X					X		34
INY	11001000	X					X		34
JMP	01X01100								33
JSR	00100000								39
LDA	101XXX01	X					X		17
LDX	101XXX10	X					X		17
LDY	101XXX00	X					X		17
LSR	010XXX10	0					X	X	37
NOP	11101010								41
ORA	000XXX01	X					X		27
PHA	01001000								40
PHP	00001000								40
PLA	01101000	X					X		40
PLP	00101000	X	X	X	X	X	X	X	40
ROL	001XXX10	X					X	X	37
ROR	011XXX10	X					X	X	38

*	BITMINTA	N	V	B	D	I	Z	C	oldalszám
RTI	01000000	X	X	X	X	X	X	X	41
RTS	01100000								39
SBC	111XXX01	X	X				X	X	24
SEC	00111000							1	35
SED	11111000				1				36
SEI	01111000					1			36
STA	100XXX01								21
STX	100XX110								21
STY	100XX100								21
TAX	10101010	X					X		22
TAY	10101000	X					X		22
TSX	10111010	X					X		22
TXA	10001010	X					X		22
TXS	10011010								22
TYA	10011000	X					X		22

Ahol a bitmintában X áll, ott a bit értéke a címzésmódtól függ. A kapcsolók alatti X arra utal, hogy a kapcsoló értéke függ az utasítás végrehajtásának eredményétől. A 0 és 1 azt jelenti, hogy a kapcsoló értéke 0, ill. 1 lesz az utasítás végrehajtása után. Ha a kapcsoló alatt nem áll semmilyen jel, az utasítás nem befolyásolja a kapcsolók értékeit.

A 6510-es utasításkódok táblázata

	0	1	2	4	5	6	8	9	A	C	D	E
0	BRK	ORA ,X)			ORA ZP	ASL ZP	PHP	ORA #	ASL A		ORA AB	ASL AB
1	BPL	ORA),Y			ORA ZPX	ASL ZPX	CLC	ORA ABY			ORA ABX	ASL ABX
2	JSR	AND ,X)		BIT ZP	AND ZP	ROL ZP	PLP	AND #	ROL A	BIT AB	AND AB	ROL AB
3	BMI	AND),Y			AND ZPX	ROL ZPX	SEC	AND ABY			AND ABX	ROL ABX
4	RTI	EOR ,X)			EOR ZP	LSR ZP	PHA	EOR #	LSR A	JMP AB	EOR AB	LSR AB
5	BVC	EOR),Y			EOR ZPX	LSR ZPX	CLI	EOR ABY			EOR ABX	LSR ABX
6	RTS	ADC ,X)			ADC ZP	ROR ZP	PLA	ADC #	ROR A	JMP IND	ADC AB	ROR AB
7	BVS	ADC),Y			ADC ZPX	ROR ZPX	SEI	ADC ABY			ADC ABX	ROR ABX
8		STA ,X)		STY ZP	STA ZP	STX ZP	DEY		TXA	STY AB	STA AB	STX AB
9	BCC	STA),Y		STY ZPX	STA ZPX	STX ZPX	TYA	STA ABY	TXS		STA ABX	
A	LDY #	LDA ,X)	LDX #	LDY ZP	LDY ZP	LDX ZP	TAY	LDA ABY	TAX	LDY AB	LDA AB	LDX AB
B	BCS	LDA),Y		LDY ZPX	LDA ZPX	LDX ZPX	CLV	CMP #	TSX	LDY ABX	LDA ABX	LDX ABX
C	CPY #	CMP ,X)		CPY ZP	CMP ZP	DEC ZP	INY	CMP ABY	DEX	CPY AB	CMP AB	DEC AB
D	BNE	CMP),Y			CMP ZPX	DEC ZPX	CLD	SBC #			CMP ABX	DEC ABX
E	CPX #	SBC ,X)		CPX ZP	SBC ZP	INC ZP	INX	SBC ABY	NOP	CPX AB	SBC AB	DEC AB
F	BEQ	SBC),Y			SBC ZPX	INC ZPX	SED				SBC ABX	INC ABX

A	akkumulátor	ZPX	nulláslap, X-szel indexelt
AB	abszolút	ZPY	nulláslap, Y-nal indexelt
ABX	abszolút, X-szel indexelt	#	közvetlen
ABY	abszolút, Y-nal indexelt	,X)	X-szel indexelt, indirekt
IND	indirekt	),Y	indirekt, Y-nal indexelt
ZP	nulláslap		